University of Alberta

Expression Data Flow Graph: Precise Flow-Sensitive Pointer Analysis for C Programs

by

Rei Thiessen

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

© Rei Thiessen
Fall 2011
Edmonton, Alberta

Abstract

Pointer analysis is a program analysis that determines the memory locations pointed to by individual pointers. Imprecise pointer information is a major impediment to data-flow analyses and back-end optimizations that depend on pointer information.

Most pointer analyses are based on a *points-to* abstraction, which is an abstraction of memory that partitions the conceptually infinite number of memory locations into a finite number of abstract objects. In a flow-sensitive pointer-analysis, a points-to relationship between abstract objects is computed at each program point.

Our pointer analysis is based on another abstraction called the *Expression Data Flow* graph, which expresses the memory dependencies between expressions that appear in a program. This abstraction represents pointer information in a more compact and more precise way than a points-to abstraction.

We present a flow-sensitive and field-sensitive algorithm that computes a precise Expression Data Flow graph of a program in a negligible amount of time.

Acknowledgements

First and foremost, I would like to thank my supervisors Dr. José Nelson Amaral and Dr. Ondřej Lhoták for their support, guidance, and patience during the course of my studies.

This work was funded by a Natural Sciences and Engineering Research Council of Canada (NSERC) scholarship and an Alberta Informatics Circle of Research Excellence scholarship.

Last but not least, I would like to thank my parents for their support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Accurate pointer information is a prerequisite to most back-end compiler optimizations and static program analyses. Without pointer information, the effects of indirect memory operations are unknown. Overly conservative assumptions of their effects reduce the effectiveness of later analyses and code optimizations. *Pointer analysis* is a static program analysis that computes pointer information.

Various types of pointer analyses have been studied for decades, but only simple pointer analyses are incorporated in today's production compilers. The reason for the sluggish uptake of advanced pointer analyses by production compilers may be unreasonable restrictions and assumptions imposed on input code by the proposed analyses. Thus, practicality in production compilers is an important design goal for new pointer analyses.

1.1 Background

Pointer analysis has been studied exhaustively, and many papers and theses have been written on this topic [12].

1.1.1 Dimensions of Pointer Analysis

Pointer analyses are categorized based on their cost/precision trade-offs. These trade-offs in pointer analyses are known as dimensions of precision. *Flow-sensitivity* and *field-sensitivity* are two major dimensions.

A flow-insensitive analysis computes one set of information that applies to all parts of a program, while a flow-sensitive analysis computes individual information for every point in a program. A *dense* flow-sensitive algorithm computes a complete set of pointer information before and after every statement. A *sparse* algorithm reduces redundant data by storing only the pointer information that has changed and only storing it at points where it has changed. An advantage of flow-sensitivity is the ability to perform *strong updates*: if the analysis can conclude that a memory location is definitely updated by an assignment, then previous values of the location can be discarded in an abstraction of pointer information.

An aggregate type in C may have multiple pointer-typed fields that may point to different objects, and pointers themselves may point to individual field members. A field-sensitive analysis differentiates both information stored in different fields and pointers pointing to different fields. In contrast, a field-insensitive analysis merges all field-specific information and maintains only whole-object information.

Most pointer analyses rely on type information provided by a compiler front-end to enable field-sensitivity. Although this reliance is reasonable for strongly-typed languages, programmers easily and commonly elude C's weak type system in real-world programs.

1.1.2 Demand-driven Analyses

Aside from the precision/performance trade-offs listed above, there are different approaches to computing pointer information. The most common approach is the *exhaustive* approach, which constructs complete points-to sets, either sparsely or densely. A *demand-driven* algorithm does not attempt this; instead, it analyzes only a portion of a program that is needed to answer a specific query.

Our approach to pointer analysis can be contrasted to previous approaches by comparing their *preprocessing cost* and *query costs*. An exhaustive analysis processes the entire program and constructs complete points-to sets. This process consumes a relatively large amount of time and memory, but all pointer information is present in the result and queries can be answered quickly. A pure demand-driven analysis performs a more rapid pass over the program and constructs a data-structure that requires significant additional computation to extract pointer and alias information.

Our approach takes the middle road between these two approaches: a program is preprocessed and some computation is performed to obtain a data-dependency graph. Recovering pointer and alias information requires some computation, but the algorithm to do so is relatively simple and fast. Furthermore, the memory footprint during the analysis and the size of the output is trivially small.

1.1.3 Unanalyzable Statements and Expressions

Most pointer analyses enforce assumptions on input programs that may hurt their applicability to real-world programs. For example, type-casts to and from pointer types may be disallowed, or analyses may require access to the source code of the whole program. If an algorithm aims to be practical in production compilers, these assumptions are fundamental flaws of the algorithm if there are no reasonable methods to eliminate them.

A major impasse is the occurrence of unanalyzable statements and expressions in a program. Examples of unanalyzable expressions include atypical pointer operations or statements that write and read back memory addresses from a file. An example of an unanalyzable statement is a procedure call to a procedure whose source code is not available to the analysis.

To conservatively handle an unanalyzable statement, an analysis must pessimistically reason that the statement may modify any address-taken variable, any variable with external linkage, or any object on the heap. For flow-insensitive algorithms, a single occurrence of an unanalyzable statement renders the pointer information of affected memory locations useless.

If the value assigned to a pointer cannot be determined, then the pointer can point to any memory location. The scalability of flow-sensitive algorithms, especially sparse algorithms, is sensitive to the size of individual points-to sets: if a pointer has numerous targets, handling dereferences of the pointer quickly becomes inefficient. Thus, handling dereferences of pointers that may point to any memory location requires special consideration.

Handling unanalyzable statements and expressions efficiently is not the only requirement: the result of the analysis in the presence of unanalyzable constructs is not useful if it is too conservative. After an unanalyzable statement, most memory locations are assumed to contain any value. In order to extract useful pointer information in such an environment, the analysis must be able to recover from this state through strong updates on memory lo-

```
1  a = &x;
2  p = &a;
3  *p = &y;
```

(a)

```
1   int g;
2
3   int main()
4   {
5       int x;
6       h(&x);
7       void* p = malloc();
8   }
9
10  void h(int* y)
11  {
12      int z;
13      if(...) h(&z);
14  }
```

(b)

Figure 1.1: Example of a C program

cations. In most flow-sensitive pointer analyses, strong updates on dynamic objects are not performed. This restriction has the unfortunate effect that pointers inside heap objects are considered to contain any value for all points in a program that follow an unanalyzable statement.

1.1.4 Points-to-set-based Pointer Analysis

The common approach to pointer analysis is based on Emami, Ghiya, and Hendren's *points-to set* abstraction, which is an abstraction of memory that maps the conceptually infinite number of memory locations to a finite number of *abstract objects* [8]. This section will provide the definitions of common program analysis terminologies and will present a brief formulation of a points-to-set-based pointer analysis. The formulation presented in this section is not a complete description of an analysis, but it is sufficient to exhibit the shortcomings of the points-to set abstraction.

C Programs

C statements are straddled by *program points*. We use the notation $\overline{N}$ and $\underline{N}$ to refer to program points immediately above and below a C statement on line N, respectively.

A *control flow graph* (CFG) is a graph representation of the control flow of a program. A node in a CFG represents a *basic block*. A basic block is a straight-line piece of code

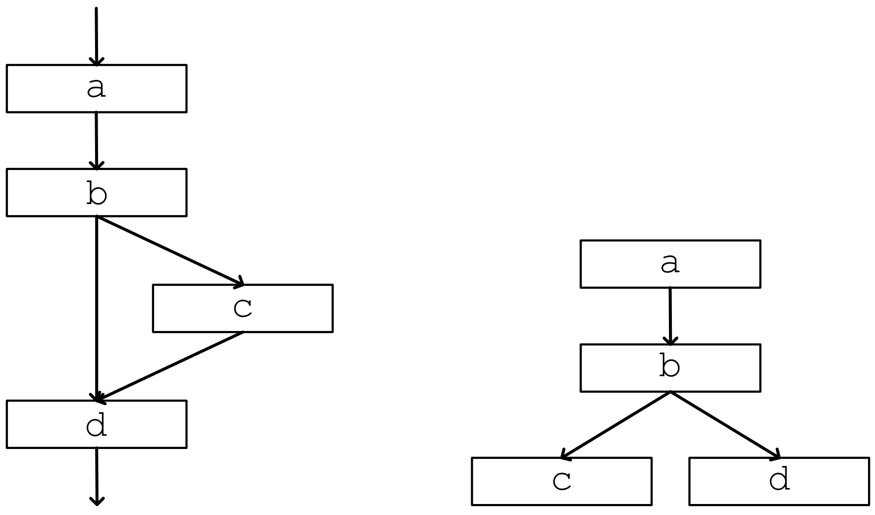


Figure 1.2: Example of a control flow graph and its dominator tree

that begins with a jump target and ends with a jump, and has no jumps or jump targets in the middle of the block. A directed edge between two basic blocks represents control flow from a jump to a jump target. An *elementary block* is a basic block that contains only one statement. An *entry block* is a block where an execution of a program begins.

A block *b dominates* b' if all control flow paths from the entry block of a program to b' passes through b. The dominator relation is similarly defined between program points, and between blocks and points. An *immediate dominator* of a block b is a block b' such that b' dominates b, and there does not exist a block $b'' \neq b'$ such that b' dominates b'' and b'' dominates b. All blocks except for the entry block have a unique immediate dominator. A *dominator tree* is a data structure where a node represents a block and an edge represents the immediate dominator relation. The root of a dominator tree is the entry block.

Figure 1.2(a) is an example control flow graph, and its dominator tree is in Figure 1.2(b). Let the block labelled "a" be the entry block. "a" is the immediate dominator of "b", and the immediate dominator of "c" and "d" is "b".

Objects are regions of memory storage. *Static objects* are allocated once at the beginning of an execution of a program. For example, in Figure 1.1(b) the object associated with the global variable g is a static object. An *automatic object* is associated with a local variable inside the scope of a procedure, and it is allocated when a procedure is entered, and deallocated when the procedure is exited. `x`, `p`, `y`, and `z` are associated with automatic objects. When the function `h` is recursively invoked, a new object is allocated and associated with `y` and `z`. When `h` is exited, `y` and `z`'s old associations are restored. Every time `malloc` is executed, it returns a newly allocated *dynamic object*.

Points-to Set Abstraction

In a points-to set abstraction, objects are mapped to *abstract objects*. Every variable is associated with an abstract object, and (concrete) objects associated with a variable are mapped to the abstract object associated with the variable. In Emami *et al.*'s paper [8], all dynamic objects are mapped to a single abstract object, but a common alternative scheme is to associate abstract objects with allocation sites, and map dynamic objects to abstract objects according to their allocation sites.

A *singular* abstract object is an abstract object that is always associated with a single concrete object at run-time. For example, abstract objects that represent global variables and local variables inside non-recursive procedures are singular abstract objects. Abstract objects that represent local variables inside recursive procedures are not singular because there may be multiple concrete objects in inactive call frames associated with a local variable in a recursive procedure.

Each program point is associated with a *points-to set*, which is a set of pairs of abstract objects. An element (p, q) in a points-to set indicates that p may point to q: an object associated with p may contain the address of an object associated with q.

A *transfer function* maps a sound abstraction of program states before a statement to a sound abstraction of program states after the statement. In a data-flow analysis, transfer functions are commonly defined through *generate* and *kill* sets: a transfer function T for a statement S on an abstraction A is defined as $T_S(A) = Gen_S(A) \cup (A \setminus Kill_S(A))$.

Let $AObj$ be the set of abstract objects. Given a points-to abstraction A and an abstract object p, let $Pts(A, p) = \{y \in AObj : (p, y) \in A\}$. Specifically, $Pts(A, p)$ is the set of points-to targets of p. Figure 1.3 lists the generate and kill sets for four types of normalized C assignment statements.

The transfer functions for statements of the form "p `= &`q`;`", "p `=` q`;`", and "p `=`

$$
\begin{array}{ll}
S = [p\ =\ \&q\,;] & Gen_S(A) = (p,q) \\
 & Kill_S(A) = \{(p,y) : y \in AObj, p \text{ is singular}\} \\
S = [p\ =\ q\,;] & Gen_S(A) = \{(p,y) : y \in Pts(A,q)\} \\
 & Kill_S(A) = \{(p,y) : y \in AObj, p \text{ is singular}\} \\
S = [p\ =\ {*q}\,;] & Gen_S(A) = \{(p,y') : y' \in AObj, \\
 & \qquad\qquad \exists y \in AObj[y \in Pts(A,q) \wedge y' \in Pts(A,y)]\} \\
 & Kill_S(A) = \{(p,y) : y \in AObj, p \text{ is singular}\} \\
S = [{*p}\ =\ q\,;] & Gen_S(A) = \{(x,y) : x \in Pts(A,p), y \in Pts(A,q)\} \\
 & Kill_S(A) = \{(x,y) : x,y \in AObj, Pts(A,p) = \{x\} \wedge x \text{ is singular }\} \\
 & \qquad\qquad \cup \begin{cases} A & \text{if } |Pts(A,p)| = 0 \\ \emptyset & \text{otherwise} \end{cases}
\end{array}
$$

Figure 1.3: Generate and kill sets

$*q$;" can kill the points-to targets of the abstract object associated with the left-hand side variable. In recursive procedures, local variables are non-singular. It may seem odd that a statement of the form "$p\ =\ q$;" does not strongly update p if p is a local variable in a recursive procedure because the statement is definitely storing to a single concrete object. However, p may represent possibly multiple concrete objects if a procedure is recursively invoked. Killing the points-to targets of p is not sound since the statement does not store to all concrete objects represented by p.

Statements of the form "$*p\ =\ q$;" are indirect stores. If p points to a single singular abstract object, then its points-to targets can be killed in an *indirect strong update*. As a data-flow analysis, the transfer function must be monotonic. In a points-to set analysis, the points-to set at a point must not decrease in size during the analysis. If a pointer does not point to any target, it may point to a single singular abstract object at a later time in the analysis, and the transfer function may kill the existing points-to relations of that target. Thus, to preserve monotonicity of the transfer function, if p does not point to any target, then the entire points-to set is killed because the transfer function may kill the points-to relations of any object at a later time in the analysis.

In Figure 1.1(a), the points-to set abstraction at $\overline{1}$ is $\{\}$; at $\underline{1} = \overline{2}$, it is $\{(a,x)\}$; at $\underline{2} = \overline{3}$, it is $\{(a,x),(p,a)\}$; and at $\underline{3}$, it is $\{(a,y),(p,a)\}$.

```
if(...) {
    p = &a;
} else {
    p = &b;
}

*p = &x;
if(...) {
    *p = &y;
}
else {
    *p = &z;
}

c = *p;
```
(a)

```
if(...) {
    p = &a;
} else {
    p = &b;
}

tmp = &x;
if(...) {
    tmp = &y;
}
else {
    tmp = &z;
}

c = tmp;
*p = tmp;
```
(b)

```
while(...) {
    if(p) {
        *p = &v;
    }

    if(...) {
        p = &a;
    } else {
        p = &b;
    }

    *p = &x;
    if(...) {
        *p = &y;
    }
    else {
        *p = &z;
        d = b;
    }

    c = *p;
}
```
(c)

Figure 1.4: Example of limitations of pointer analysis

Limitations

The code listing in Figure 1.4(a) is used to illustrate imprecision in a points-to-set-based analysis. If we perform a manual analysis of the code, we can conclude that the variable c may point to either y or z. In a points-to-set-based analysis, none of the stores through *p are strong updates because p may point to two targets. Thus, a points-to-set-based analysis concludes that c may point to x, y, or z.

Figure 1.4(b) is a code listing where a simple variable substitution is applied to Figure 1.4(a). The occurrences of *p are substituted with a temporary variable. In this case, the transformation is sound, and a points-to-set-based analysis is precise on the transformed code. Figure 1.4(c) is a modification of Figure 1.4(a) that complicates performing the same variable substitution on the code:

- A segment of the code is inside a loop, and the pointer variable p is redefined in every iteration.

- Inside the loop, the value of p that was defined in a previous iteration is used in an indirect store (in line 3), and the value of p in the previous iteration may be different from the value of p in the current iteration.

- There is a use of the variable b (in line 18), and stores to *p may affect the values stored in b. Variable substitutions must account for this behaviour. Aliasing between expressions cannot be determined without performing pointer analysis, and thus, pointer analysis must preclude the variable substitution transformation.

Through a manual analysis of Figure 1.4(c), the points-to targets of c is still y and z. If an analysis, aiming to improve its precision, performs variable substitution only when complications, such as the ones listed above, do not appear, then the resulting analysis will be *brittle*, meaning that small modifications to the code result in large changes to the precision of the output. One of our goals is to design a *flexible* analysis that is more precise than a points-to-set-based pointer analysis.

1.2 Contributions

Previous sections have described open problems in existing pointer analyses. This thesis addresses these problems with the following contributions:

- We present an efficient representation of pointer information. The representation avoids decomposing indirect memory operations into their effects on variables, and thus the size of the representation is closer to the size of the input.

- The representation is designed to efficiently handle unanalyzable constructs, and our algorithm avoids using type information. The avoidance of type information allows our analysis to soundly handle non-portable C constructs that appear in real-world programs. In effect, our analysis requires only a low-level, assembly-like representation of code.

- We present an algorithm that computes the proposed representation using a negligible amount of time and space. The precision of pointer information in the output exceeds that of traditional flow-sensitive pointer analyses.

1.3 Mathematical Notation

The power-set of a set S is written $\mathcal{P}(S)$. Given a set S, the application of the Kleene operator on S is written S^*.

The set-builder notation appears in this document in two forms: $\{x \in S : P(x)\}$ or $\{F(x_1, x_2, \ldots) : x_1 \in S_1, x_2 \in S_2, \ldots, P(x_1, x_2, \ldots)\}$, where F is a function and P is a propositional formula.

Sequences are enclosed in square brackets. Concatenation of sequences is denoted by juxtaposition: $[1, 1, 1] \cdot [2, 2] \equiv [1, 1, 1, 2, 2]$.

Given $\mu : \mathbf{X} \to \mathbf{Y}$, $x \in \mathbf{X}$, and $y \in \mathbf{Y}$, $\mu[x \mapsto y]$ is a transformation of a mapping μ where $\mu[x \mapsto y](x) = y$ and $\forall z \neq x, \mu[x \mapsto y](z) = \mu(z)$. $\mu[\{x_1, x_2, \ldots\} \mapsto y]$ is equivalent to $\mu[x_1 \mapsto y, x_2 \mapsto y, \ldots]$.

Given $\mu : \mathbf{X} \to \mathcal{P}(\mathbf{Y})$, $x \in \mathbf{X}$, and $Y \in \mathcal{P}(\mathbf{Y})$, $\mu[x \mapsto^+ Y]$ is a transformation of a mapping μ where $\mu[x \mapsto^+ Y] = \mu[x \mapsto \mu(x) \cup Y]$. $\mu[\{x_1, x_2, \ldots\} \mapsto^+ Y]$ is equivalent to $\mu[x_1 \mapsto^+ Y, x_2 \mapsto^+ Y, \ldots]$.

1.4 Document Organization

Chapter 2 describes the representation of a program that our pointer analysis expects as input. Chapter 3 describes the output representation and the flow-sensitive and field-sensitive pointer analysis algorithm. Before the flow- and field-sensitive algorithm is described, a flow-insensitive algorithm is presented to familiarize readers with our unusual abstraction of a program. Chapter 4 contains the experimental evaluations. Chapter 5 examines related works. Chapter 6 is the conclusion of this thesis.

Terms that have a specific meaning attached to it and constants, variables, and functions that are referenced in multiple sections, are defined in a labelled definition.

Chapter 2

Concrete Semantics

An Expression Data Flow (EDF) graph is an abstraction of a program that represents data dependencies between expressions. Most data-flow graphs are variable-centric in that they represent indirect memory operations by their effects on variables: the left- and right-hand side expressions of an assignment statement are decomposed into sets of variables modified by the assignment (variables aliased with the left-hand side expression) and the set of variables loaded by the assignment (variables aliased with the right-hand side expression). In contrast, the EDF graph expresses data dependencies directly between expressions that appear in a program, which results in a smaller and more precise representation.

When a statement "`*p = *q;`" is executed, a typical list of low-level operations performed in the execution is listed below:

1. Load from the memory location addressed by `&q` (a constant) into register r_1.
2. Load from the memory location addressed by r_1 into register r_2.
3. Load from the memory location addressed by `&p` into register r_3.
4. Store the value of r_2 to the memory location addressed by r_3.

There are three loads performed in the above assignment statement. Whenever a value is loaded from a memory location, the EDF graph relates the load to the store operation that was last to store to that memory location.

This chapter presents a *concrete semantics* for a representative instruction set. A concrete semantics precisely and intuitively models all possible runtime states of a program. It cannot be computed in finite time and space for all programs, but it provides an understanding of the behaviour of an abstract machine that operates on our representative instruction set.

2.1 Concrete Semantics

Most formulations of the semantics of a programming language are divided into the semantics of an *evaluation* of an expression, which produces a value, and an *execution* of a statement, which modifies a program state. The construction of an EDF graph is performed in increments that correspond to each step of an evaluation of an expression in addition to an execution of a statement. We describe an intermediate language whereby every individual memory access involved in evaluating a complex expression is represented as a distinct step of computation, which matches the nature of our abstraction more closely than a statement-based programming language.

The concrete semantics models an abstract machine with an infinite memory space. The memory space is partitioned into *objects*. A *memory location* is an unit of memory where a single *value* can be stored and from where that value can be retrieved at a later time in an execution of a program. A value is either the address of a memory location or a symbol representing an uninitialized value. For the purpose of the concrete semantics, non-address-typed values stored in memory locations are not of interest. Memory locations are addressed by an integer *offset* within a particular object. All non-negative integers are valid offsets and thus each object contains an infinite number of memory locations.

In the concrete semantics, the memory spaces of objects are disjoint. In real-world machines, objects are allocated in a flat memory space and memory locations inside one object can be accessed through pointer arithmetic on an address of a different object. Our concrete semantics does not model programs containing such operations because the C standard does not define a behaviour for them.

Let Obj be a set of objects. Let $MLoc = Obj \times \mathbb{Z}$ be a set of memory locations, where $(o, f) \in MLoc$ is interpreted as an offset f inside an object o. An element $x \in MLoc$ can be interpreted as a memory location x or, if it is used as a value, it can be interpreted as the address of a memory location x. Let $Val = MLoc \cup \{\epsilon\}$ be a set of values that may

$$\begin{aligned}
\mathit{Instr} &= [\![\mathit{InstrType}]\!]^{\mathit{ILabel}} \\
\mathit{InstrType} &= \text{“LOADA”}\ R\ \mathit{Var} \\
&\mid \text{“LOADM”}\ R \\
&\mid \text{“LOAD”}\ R\ \mathbb{Z} \\
&\mid \text{“STORE”}\ \mathbb{Z}\ \mathbb{Z} \\
&\mid \text{“SKIP”}
\end{aligned}$$

Figure 2.1: Instruction set grammar

be stored in a memory location, where a member that is a memory location is interpreted as the address of the memory location, and ϵ denotes an uninitialized value. Addition and subtraction between an element of *MLoc* and an integer is carried out on the offset of the memory location: $(o, f) + m \equiv (o, f + m)$.

Every variable in a program is associated with a unique object. Objects can also be allocated by a dynamic allocation function. Allocating an object means that each execution of an allocation function returns an object unique to that execution.

Definition 2.1. Figure 2.1 is a grammar of our representative instruction set. Let *ILabel* and *Var* be sets of terminals. Each instruction in a program is uniquely labelled by a label from the set *ILabel*. The label of an instruction is used to refer to the instruction itself. *Var* is a set of identifiers of local variables. Let $R = \{r_L, r_R\}$ be a set of terminals used to select the two registers of the abstract machine that operates on the instruction set: r_L and r_R. The registers r_L and r_R are used to hold intermediate values of executions of instructions that represent the left-hand side and the right-hand side of an assignment statement, respectively.

Let $r \in R$, $v \in \mathit{Var}$, and $f, m \in \mathbb{Z}$. “LOADA r v” writes the address associated with v to register r. “LOADM r” allocates an object and writes its address to register r. “LOAD r f” reads a value x from register r, loads a value y from the memory location addressed by $x+f$, then writes y to r. f is called the instruction’s *dereference offset*, or *d-offset*. “STORE f m” reads a value x from register r_L, reads a value y from register r_R, and then stores the value $y + m$ into the memory location addressed by $x + f$. f is the instruction’s d-offset, and m is called the instruction’s *modifier*. SKIP instructions do not modify the program state and they are used as placeholders.

When referring to a LOADA, LOADM, or LOAD instruction, the term LOAD(AM) is used. LOADA and LOADM instructions are *root instructions* because they generate a value without loading from memory. A *def-instruction* is an instruction that stores to a memory location, and a *use-instruction* is either a root instruction or an instruction that loads from a memory location. STORE instructions are def-instructions, while LOAD(AM) instructions are use-instructions.

Definition 2.2. A LOAD(AM) instruction is said to *produce* the value that it writes to a register when executed. A STORE instruction produces the value it stores to memory when executed.

Utility functions that return local information about an instruction are defined below.

Definition 2.3. A LOAD or STORE instruction dereferences an address written to a register by a preceding instruction. Let l be a LOAD or STORE instruction and let l_s be the LOAD(AM) instruction that produces the value that is dereferenced by l. l_s is called the *dereference source (d-source)* of l. Let $DSrc : ILabel \rightarrow ILabel$ map a LOAD or STORE instruction to its d-source instruction.

For example, if there is a control flow edge from "$[\![\text{LOAD } r_R\ 0]\!]^{l_1}$" to "$[\![\text{LOAD } r_R\ 0]\!]^{l_2}$" then l_1 is the dereference source of l_2; if there is a control flow edge from $[\![\text{LOAD } r_L\ 0]\!]^{l_3}$ to $[\![\text{STORE } 0\ 0]\!]^{l_4}$ then l_3 is the dereference source of l_4.

Definition 2.4. A STORE instruction stores a value, written to the register r_R by a preceding instruction, to a memory location. Let l be a STORE instruction and let l_v be the LOAD(AM) instruction that wrote the value that is stored by l, to the register r_R. l_v is called the *store-value instruction* of l. Let $StVal : ILabel \rightarrow ILabel$ map a STORE instruction to its store-value instruction.

For example, if there is a control flow edge from "$[\![\text{LOAD } r_R\ 0]\!]^{l_1}$" to "$[\![\text{STORE } 0\ 0]\!]^{l_2}$" then l_1 is the store-value instruction of l_2.

Definition 2.5. A LOAD or STORE instruction adds an offset (possibly zero) to the address produced by its d-source instruction to generate a different address. This offset is called

```
1  p = &a;
2  *(p + 1) = &x + 2;
3  b = *(p + 1) + 3;
```

(a) Code

```
 1  SKIP
 2  LOADA r_R a
 3  LOADA r_L p
 4  STORE 0 0
 5  LOADA r_R x
 6  LOADA r_L p
 7  LOAD  r_L 0
 8  STORE 1 2
 9  LOADA r_R p
10  LOAD  r_R 0
11  LOAD  r_R 1
12  LOADA r_L b
13  STORE 0 3
```

(b) Listing

Figure 2.2: Example of concrete semantics

the *dereference offset (d-offset)* of *l*. Let *DOff* : *ILabel* $\rightarrow \mathbb{Z}$ map LOAD and STORE instructions to their d-offsets.

Definition 2.6. Let *Modf* : *ILabel* $\rightarrow \mathbb{Z}$ map STORE instructions to their modifiers.

For simplicity, we assume that d-offsets and modifiers are constant integers in this chapter. Real-world programs may perform pointer addition and subtraction with non-constants. Section 3.8 on field-sensitivity introduces abstractions that can represent non-constant d-offsets and modifiers.

Figure 2.2(a) is an example C program, and Figure 2.2(b) is a transformation of the C program into our representative instruction set.

2.2 C Programs

Assignment statements are the only type of C statements that change the representation of a program state. The C expressions that we model can be parsed as a series of *sub-expressions* nested by the dereference operator. For example, `*(p + 1)` has three sub-expressions: `&p`, `*(&p)`, and `*(*(&p)+1)`. Each sub-expression corresponds to an instruction. The *base sub-expression* is the address-of expression. If every line has no more than one assignment statement, then sub-expressions, and thus instructions, can be referenced unambiguously using pairs of line numbers and sub-expressions. Let `k:L(`e`)` refer to a sub-expression e

in the left-hand side expression of a statement on line k of a C program. Similarly, let k:R(e) refer to a sub-expression e in the right-hand side expression of a statement on line k. This notation is used for labels of instructions. Line numbers of C program listings serve as *statement labels*. For example, the instruction "STORE 0 0" on line 4 in Figure 2.2(b) corresponds to `1:L(p)` in Figure 2.2(a). Thus, the labelled instruction is written $[\![\text{STORE } 0\ 0]\!]^{1:\mathtt{L}(p)}$. Likewise, "LOAD r_R 1" on line 11 corresponds to $3{:}\mathtt{R}(*(p+1))$ and the labelled instruction is $[\![\text{LOAD } r_R\ 1]\!]^{3:\mathtt{R}(*(p+1))}$.

A LOADM instruction is labelled &k, where k is the statement label of the statement containing the LOADM instruction. For example, if there is a C statement "`p = malloc();`" on line 3 of a program, then the labelled instruction representing the right-hand-side expression is $[\![\text{LOADM } r_R]\!]^{\&3}$.

A complete programming language requires conditional branch instructions, where the next instruction to execute is determined by the program state at the point of the branch instruction. A flow-sensitive program analysis that uses its abstraction of the program state to determine which branch target may or may not be taken is called a *path-sensitive* analysis. Our analysis is path-insensitive. Thus, the control flow of programs can be abstracted as a mapping from one instruction to a set of possible succeeding instructions, where jumping to any one of the succeeding instructions is considered to be a valid sequence of execution.

To simplify the presentation of our analysis, we use the following assumptions:

- The blocks of the control flow graph are elementary blocks, which are blocks that contain exactly one instruction. There is a program point *above* and *below* every instruction.
- The entry block is *isolated*, meaning there are no control flow edges that target the entry block.
- The entry block is a SKIP instruction.
- If a block has multiple blocks that immediately precede it, then the block is a SKIP instruction.

Definition 2.7. Let $\mathit{CFlow} : \mathit{ILabel} \times \mathit{ILabel}$. Given instructions l and l', let $(l, l') \in \mathit{CFlow}$

if there is a control flow edge from l to l'.

Definition 2.8. Let $l_{entry} \in ILabel$ be the entry block of a program. All program points in a program are dominated by l_{entry}. We assume that l_{entry} labels a SKIP statement.

Definition 2.9. Let *ProgPt* be the set of program points. There is a program point above and below every instruction. Given an instruction l, let $PtAbove(l)$ be the point above l, and let $PtBelow(l)$ be the point below l.

Definition 2.10. An execution trace is a sequence of instructions that represent a possible execution of a program. Traces are elements of $ILabel^*$. The value *produced by an execution trace* is the value produced by the execution of the trace's last instruction when the trace is executed. An execution trace *ends at a point t* if t is below the last instruction in the trace.

The following definitions are utility functions that operate on traces.

Definition 2.11. Given a trace tr, let $Last(tr)$ be the last instruction in tr.

Definition 2.12. Given a trace tr and an instruction l, let $LPrefix(tr, l)$ be the longest prefix of tr that ends with l.

2.3 Program State

Recording the values stored in memory locations is sufficient to describe an abstraction based on points-to sets. The EDF graph abstraction requires higher-level information about a program: in addition to recording values stored in memory locations, the concrete semantics records the last instruction that stored to each memory location.

Definition 2.13. Let *VState* be the set of program states:

$$\begin{aligned} VState = {} & (MLoc \to Val) \\ & \times (MLoc \to (ILabel^* \cup \{\epsilon\})) \\ & \times (R \to Val) \\ & \times ILabel^* \end{aligned}$$

Given a program state $(\sigma, \mu, \rho, tr) \in \mathit{VState}$, the components of the state are described below:

- σ is a mapping from memory locations to values stored in them. If a memory location x is uninitialized, then $\sigma(x) = \epsilon$.
- μ is a mapping from a memory location to the execution trace up to the execution of an instruction that last stored to the memory location. Given a memory location x, the instruction $\mathit{Last}(\mu(x))$ is the *Most Recent Update (MRU)* of x. If a memory location x is uninitialized, then $\mu(x) = \epsilon$.
- ρ is a mapping from registers to values stored in them.
- tr is an execution trace which begins with the first instruction of a program and which was executed and resulted in the state (σ, μ, ρ, tr).

The set of objects Obj is defined as $\mathit{Obj} = \mathit{Var} \cup \mathit{ILabel}^*$: an object is either uniquely associated with a variable or it is uniquely associated with an execution trace. Static and automatic objects are associated with variables. A dynamic object is associated with the execution trace up to the execution of a LOADM instruction that allocated the object. Within an execution trace, each invocation of a dynamic allocation function returns a unique object.

Definition 2.14. The initial state V_0 at the entry point of a program is defined as $\mathit{VState}_0 = (\lambda x.\epsilon, \lambda x.[], \lambda r.\epsilon, [])$.

Definition 2.15. Figure 2.3 lists the semantics of instructions. A state transition is denoted $I \vdash V \rightsquigarrow V'$, where I is a labelled instruction and V' is the resulting state after executing I on the state V. Alternatively, the transition may also be written $l \vdash V \rightsquigarrow V'$ where l is the label of an instruction.

When a "LOADA r v" instruction is executed, the address of the memory location $(v, 0)$ is written to register r. When a "LOADM r" instruction is executed, the address within an object that is unique to that execution trace is written to register r. When a "LOAD r f" instruction is executed, the value stored in the memory location $\rho(r) + f$ is written to register r. When a "STORE f m" instruction is executed, the value stored in the memory location

$$
\begin{array}{rcll}
[\![\text{LOADA } r\ v]\!]^l & \vdash (\sigma,\mu,\rho,tr) \rightsquigarrow & (\sigma,\mu,\rho[r \mapsto (v,0)], tr \cdot [l]) & \\
[\![\text{LOADM } r]\!]^l & \vdash (\sigma,\mu,\rho,tr) \rightsquigarrow & (\sigma,\mu,\rho[r \mapsto (tr,0)], tr \cdot [l]) & \\
[\![\text{LOAD } r\ f]\!]^l & \vdash (\sigma,\mu,\rho,tr) \rightsquigarrow & (\sigma,\mu,\rho[r \mapsto \sigma(\rho(r)+f)], tr \cdot [l]) & \text{if } \rho(r_R) \neq \epsilon \\
[\![\text{STORE } f\ m]\!]^l & \vdash (\sigma,\mu,\rho,tr) \rightsquigarrow & (\sigma[(\rho(r_L)+f) \mapsto (\rho(r_R)+m)], & \\
 & & \mu[(\rho(r_L)+f) \mapsto tr \cdot [l]], \lambda r.\epsilon, tr \cdot [l]) & \text{if } \rho(r_L) \neq \epsilon \\
[\![\text{SKIP}]\!]^l & \vdash (\sigma,\mu,\rho,tr) \rightsquigarrow & (\sigma,\mu,\rho, tr \cdot [l]) &
\end{array}
$$

Figure 2.3: Instruction semantics

$\rho(r_R) + m$ is stored to the memory location $\rho(r_L) + f$, and the MRU of $\rho(r_L) + f$ is set to the STORE instruction.

Definition 2.16. Let $tr \vdash V$ if V is the resulting state after executing the trace tr beginning with the initial state V_0. We say that V is the *state reached by tr*, or *tr leads to* the state V.

Not all traces lead to a state: when an uninitialized value is dereferenced in a trace, then the trace does not lead to a program state. A static analysis computes properties of program states reached by execution traces. Thus, an analysis does not have to consider execution traces that dereference an uninitialized value.

For this example, line numbers in Figure 2.2(b) serve as instruction labels. Let (σ, μ, ρ, tr) be the state reached by the trace $tr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$. The values of the components are listed below:

$$\sigma = \{((\mathtt{a}, 1), (\mathtt{x}, 2)), ((\mathtt{p}, 0), (\mathtt{a}, 0))\}$$

$$\mu = \{((\mathtt{a}, 1), [1, 2, 3, 4, 5, 6, 7, 8]), ((\mathtt{p}, 0), [1, 2, 3, 4])\}$$

$$\rho = \{(r_L, (\mathtt{b}, 0)), (r_R, (\mathtt{x}, 2))\}$$

$$tr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]$$

The tuple $((\mathtt{a}, 1), (\mathtt{x}, 2))$ in σ represents a mapping from the memory location $(\mathtt{a}, 1)$ to the value $(\mathtt{x}, 2)$. The MRU of $(\mathtt{a}, 1)$ is 8, and the MRU of $(\mathtt{p}, 0)$ is 4.

The next chapter describes a representation of a program as an EDF graph and presents algorithms to compute the EDF graph. The graph represents instructions as nodes. Instead of recording MRUs of individual memory locations, an EDF graph relates an instruction that loads a memory location to the MRU of the memory location with an edge between the two nodes representing the two instructions. This relation is called a *reaching definition*. For example, in Figure 2.2(b), a node that represents the instruction labelled 11 loads from the memory location $(\mathsf{a}, 1)$. The MRU of $(\mathsf{a}, 1)$ at the point of the load is the instruction labelled 8. Thus, the node that represents the instruction 8 is a reaching definition to the node that represents the instruction 11.

Chapter 3

Abstraction

The EDF graph is a sparse and approximate abstraction of the concrete semantics. A high-level description is one such that a node of an EDF graph represents an instruction, and edges relate load instructions to store instructions such that if, in some execution, a load instruction l loads from a memory location x, then an edge is present between the node that represents x's MRU and the node that represents l. Points-to sets can be derived from the graph through graph reachability queries.

The EDF graph is introduced through two abstraction domains. The first abstraction domain introduces a simplified EDF graph. A flow-insensitive algorithm that computes an abstraction in the first abstraction domain is presented.

Computing a precise abstraction in the first abstraction domain is difficult. The second abstraction domain is the "proper" EDF graph and an algorithm that computes a precise abstraction is defined in the second abstraction domain.

3.1 Abstraction Domain *Graph*

Definition 3.1. The abstraction domain *Graph* is defined below.

$$Graph = \mathcal{P}(Node) \times \mathcal{P}(Edge)$$

$$Node = ILabel \cup Var$$

$$Edge = Node \times Node$$

Let *Graph* be partially ordered by $\sqsubseteq$:

$$(G_N, G_E) \sqsubseteq (G'_N, G'_E) \equiv G_N \subseteq G'_N \wedge G_E \subseteq G'_E$$

An element of *Graph* is an *abstraction*. Given $G = (G_N, G_E) \in \mathit{Graph}$, G_N is a set of nodes. A node corresponds to an instruction. Except for LOADA instructions, instructions map one-to-one to nodes. G_E is a set of edges. An edge from a node n' to n indicates that n' is a possible reaching definition of n: in some execution of a program, the instruction represented by n may load a memory location whose MRU is the STORE instruction represented by n'.

LOADA and LOADM instructions are root instructions because they are the source of addresses for chains of memory dependencies. Root instructions are represented by *root nodes* such that the set of root nodes represents a partition of the memory space. Specifically, if executions of two root instructions may produce the same value, then they are mapped to the same root node. LOADM instructions produce a unique value in every execution; thus each LOADM instruction is represented by a unique node. LOADA instructions of the form "LOADA r v" produce a constant value that is unique to the variable v; thus all instances of a "LOADA r v" instruction for a particular variable v, are represented by a unique root node associated with the variable v.

Nodes that represent LOADA instructions are *address nodes*. Nodes that represent LOADM instructions are *malloc nodes*. *Store nodes* abstract STORE instructions, and *load nodes* abstract LOAD(AM) instructions.

Definition 3.2. The *dereference source of a node n* or *d-source* of n, is the node that represents the d-source of the instruction represented by n. Root nodes are the only nodes that do not have a d-source. The *dereference offset of a node n* or *d-offset* of n, is the d-offset of the instruction represented by n. If the d-source of n is n_s and the d-offset of n is f, then we may use the phrase, "the node n that is d-sourced from n_s with d-offset f," to refer to n.

Definition 3.3. The mapping between instructions and their representative nodes, *ItoN*, is defined below:

$$\mathit{ItoN} : \mathit{ILabel} \rightarrow \mathit{Node}$$

$$\mathit{ItoN}(l) = \begin{cases} v & \text{if } l \text{ labels LOADA } v \text{ where } v \in \mathit{Var} \\ l & \text{otherwise} \end{cases}$$

In Figure 2.2(b), labels 2 and 3 label LOADA instructions and the label 4 labels a STORE instruction. Therefore, $\mathit{ItoN}(2) = \texttt{a}$, $\mathit{ItoN}(3) = \texttt{p}$, and $\mathit{ItoN}(4) = 4$.

Definition 3.4. With the exception of address nodes, the *position* of a node in the context of a program's control flow graph is the instruction represented by the node. The position of an address node is the entry instruction.

When discussing graph properties, there may be confusion between the control flow graph of a program and the abstraction graph. The unqualified term "nodes" is reserved for nodes of an abstraction. We assume that the control flow graph of an input program consist of elementary blocks; thus the nodes of a control flow graph are instruction. To avoid ambiguities between abstraction graphs and control flow graphs, we explicitly qualify graph terms such as "edges" and "paths" as *data flow edges* (df-edges) and *data flow paths* (df-paths) when they refer to the abstraction, and *control flow edges* (cf-edges) and *control flow paths* (cf-paths) when they refer to the control flow graph. A node n is *df-reachable* from a node n' if there exists a df-path from n to n'. A node n is *cf-reachable* from a node n' if there exists a cf-path from n's position to n''s position. A node n dominates a node n' if n's position dominates n''s position. A node dominates a point if the position of the node dominates the point, and like-wise for points dominating nodes.

3.2 Slices

An abstraction abstracts a set of execution traces. Informally, a property derived from a trace is realized as properties in the abstract domain. In our abstraction, the property of interest that is derived from a trace is a concept of a chain of memory dependencies, called a *slice* [26]. The slice of a trace is the history of loads and stores of the value produced by the trace.

Definition 3.5. A trace is associated with a slice. The function *Slice* defined in Figure 3.1 constructs a slice from a trace.

Given a trace, the trace's slice is constructed from the end of the trace to the beginning of the trace. The first case in the piecewise definition handles the case when the trace does not lead to a program state: this occurs when an uninitialized value is dereferenced somewhere in the trace. The second case states that the slice of an execution of a root instruction is just the root instruction: there is no memory dependence to any other part of a program for the

Let $tr, tr' \in ILabel^*, l \in ILabel, (\sigma, \mu, \rho, tr) \in VState, r, f \in \mathbb{Z}$.

$$Slice : ILabel^* \to ILabel^*$$

$$Slice(tr \cdot [l]) = \begin{cases} [] & \text{if } \nexists V \in VState \\ & \quad \mid tr \vdash V \\ [l] & \text{if } l \text{ labels a root instruction} \\ Slice(tr') \cdot [l] & \text{if } l \text{ labels LOAD } r\ f \\ & \quad \text{where } tr \vdash (\sigma, \mu, \rho, tr) \\ & \quad \text{and } tr' = \mu(\rho(r) + f) \\ Slice(LPrefix(tr, StVal(l))) \cdot [l] & \text{if } l \text{ labels a STORE instruction} \end{cases}$$

Figure 3.1: Slice function

instruction to produce a value. The third case uses the μ mapping to determine the MRU of the loaded memory location $\rho(r) + f$. The fourth case ties a store instruction l that stores a value to the load instruction $StVal(l)$ that loaded the value.

An abstraction abstracts a trace if the trace's slice is represented in the graph. Traces and slices are sequences of instructions, and they can be mapped to df-paths in the abstraction through *ItoN*.

Definition 3.6. Given a sequence of instructions S, let $NodeSeq(S)$ be the element-wise map of S by *ItoN*. Given a trace *tr*, we refer to $NodeSeq(tr)$ as the *trace path* of *tr*, and $NodeSeq(Slice(tr))$ as the *slice path* of *tr*[1].

$$NodeSeq : ILabel^* \to Node^*$$

$$NodeSeq(tr) = [ItoN(l_1), \ldots, ItoN(l_i)] \qquad \text{where } [l_1, \ldots, l_i] = Slice(tr)$$

With the exception of slices that consist of a single root instruction, slices correspond one-to-one to slice paths. If a slice path is represented in an abstraction, then every adjacent pair of nodes in a slice path must be an edge in the graph. Thus, we arrive at an *abstraction function* that maps a set of traces to the most precise abstraction in *Graph* that abstracts the set.

[1]If we follow the guidelines for avoiding control-flow/data-flow ambiguities stated in the previous section, then the two terms should be *trace df-path* and *slice df-path*, but we disregard the guidelines for these two terms for brevity.

Definition 3.7. The abstraction function for *Graph*, α, is defined below. α is defined component-wise.

$$\begin{aligned}
\alpha &: \mathcal{P}(\mathit{ILabel}^*) \rightarrow \mathit{Graph} \\
\alpha(H) &= (\alpha_N(H), \alpha_E(H)) \\
\alpha_N(H) &= \{n \in \mathit{Node} : (\exists tr \in H \mid\ l \text{ appears in } \mathit{NodeSeq}(tr))\} \\
\alpha_E(H) &= \{(n, n') : n, n' \in \mathit{Node}, \\
&\qquad (\exists tr \in H \mid n \text{ is immediately followed by } n' \text{ in } \mathit{NodeSeq}(\mathit{Slice}(tr)))\}
\end{aligned}$$

Given a set of traces $H \in \mathcal{P}(\mathit{ILabel}^*)$ and an abstraction $G \in \mathit{Graph}$, if $\alpha(H) \sqsubseteq G$, then G abstracts H. The goal of the analysis is to compute an abstraction that abstracts all traces of a program.

Definition 3.8. Let $\mathit{CS} \subseteq \mathit{ILabel}^*$ be the *collecting semantics*. The collecting semantics records all possible traces of a program. CS is defined below:

$$\begin{aligned}
f(H) &= \{[l_{entry}]\} \cup \{tr \cdot [l] : tr \in H, (\mathit{Last}(tr), l) \in \mathit{CFlow}\} \\
\mathit{CS} &= \bigcap \{f(H) = H\}
\end{aligned}$$

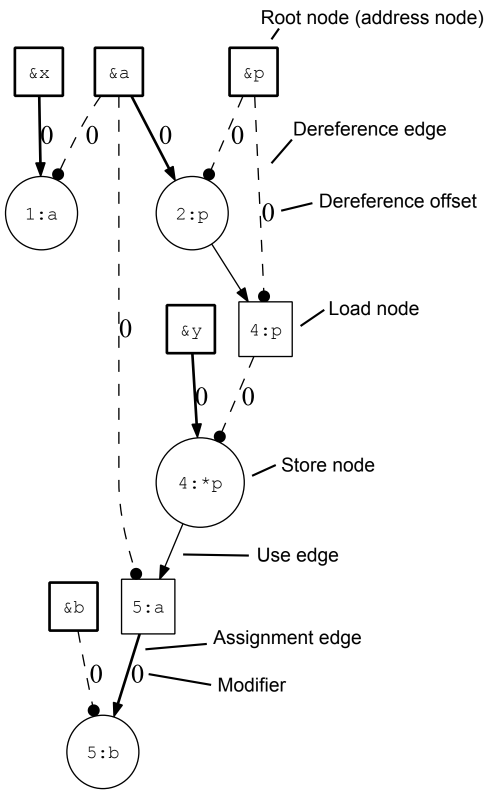
Given a set of traces H, $f(H)$ is a set that contains the trace that consists of the first instruction of a program and traces in H extended by succeeding instructions in the control flow graph. CS is the least fixed point of f, and it is the smallest set that contains all execution traces of a program.

Definition 3.9. An abstraction $G \in \mathit{Graph}$ *abstracts a program* if it abstracts all traces of a program: $\alpha(\mathit{CS}) \sqsubseteq G$.

Figure 3.2(b) is an abstraction of the code in Figure 3.2(a). A node is labelled with the label of its associated instruction. Square nodes are load nodes and circle nodes are store nodes. Thin solid edges are *use edges* (an edge from a store node to a load node) and bold solid edges are *assignment edges* (an edge from a load node to a store node). Dotted edges are *dereference edges*; a dereference edge indicates that the source of the edge is the d-source node of the destination node. A number alongside a dereference edge is the d-offset

```
1  a = &x;
2  p = &a;
3
4  *p = &y;
5  b = a;
```

(a) Code



(b) Graph

Figure 3.2: Example of an abstraction

of the edge's destination node. A number alongside an assignment edge is the modifier of the edge's destination node.

3.3 Flow-insensitive and Field-insensitive Algorithm

This section presents a trivial algorithm that computes an imprecise abstraction of a program. The precision of the analysis is similar to the precision of a flow-insensitive and field-insensitive pointer analysis.

The input to the algorithm is a collection of various sets and mappings that describe a program[2]. They were introduced in the previous chapter, but they are listed here for reference:

- *ILabel* is the set of labels of instructions.
- *Instr* is the set of labelled instructions.
- $CFlow \in \mathcal{P}(ILabel \times ILabel)$ is the set of control flow edges between instructions.
- *Var* is the set of variables.
- $StVal : ILabel \rightarrow ILabel$ is the mapping from a STORE instruction to its store-value instruction.
- $DSrc : ILabel \rightarrow ILabel$ is the mapping from a STORE or LOAD instruction to its d-source instruction.
- $DOff : ILabel \rightarrow \mathbb{Z}$ is a mapping from a STORE or LOAD instruction to its d-offset.
- $Modf : ILabel \rightarrow \mathbb{Z}$ is a mapping from a STORE instruction to its modifier.
- l_{entry} is the first instruction of the program.

The output of the algorithm is an abstraction of the program.

[2] *StVal* and *DSrc* can be derived from *ILabel* and *CFlow* by performing a single pass over a program's control flow graph

3.3.1 Transfer Functions

Given a labelled instruction $[\![i]\!]^l$, if $G \in \mathit{Graph}$ abstracts a set of traces H, let the transfer function $\mathit{TsfrInstr} : \mathit{Graph} \times \mathit{Instr} \rightarrow \mathit{Graph}$ be defined such that $\mathit{TsfrInstr}(G, [\![i]\!]^l)$ abstracts $H \cup \{tr \cdot [l] : tr \in H,\ (\mathit{Last}(tr), l) \in \mathit{CFlow}\}$. The transfer function is defined component-wise by sets of nodes and edges, and piece-wise by instruction type: $\mathit{TsfrInstr}(G, I) = (\mathit{TsfrInstr}_N(G, I), \mathit{TsfrInstr}_E(G, I))$.

The definition of the transfer function for a SKIP instruction is $\mathit{TsfrInstr}(G, [\![\text{SKIP}]\!]^l) = G$.

The transfer function for a LOADA or LOADM instruction is trivial: a slice of a trace ending with a root instruction is a list consisting of just the root instruction. The only changes required to an abstraction is the addition of a node that represents the root instruction.

Let $l \in \mathit{ILabel}, r \in R, v \in \mathit{Var}, G \in \mathit{Graph}$.

$$\mathit{TsfrInstr}_N(G, [\![\mathit{LOADA}\ r\ v]\!]^l) = G_N \cup \{\mathit{ItoN}(l)\}$$
$$\mathit{TsfrInstr}_E(G, [\![\mathit{LOADA}\ r\ v]\!]^l) = G_E$$

$$\mathit{TsfrInstr}_N(G, [\![\mathit{LOADM}\ r]\!]^l) = G_N \cup \{\mathit{ItoN}(l)\}$$
$$\mathit{TsfrInstr}_E(G, [\![\mathit{LOADM}\ r]\!]^l) = G_E$$
$$\text{where } G = (G_N, G_E)$$

The transfer function for a STORE instruction is also trivial: from the definition of *Slice*, we can observe that, in a slice, the instruction that comes before a STORE instruction l is always $\mathit{StVal}(l)$.

$$\mathit{TsfrInstr}_N(G, [\![\mathit{STORE}\ f\ m]\!]^l) = G_N \cup \{\mathit{ItoN}(l)\}$$
$$\mathit{TsfrInstr}_E(G, [\![\mathit{STORE}\ f\ m]\!]^l) = G_E \cup \{(\mathit{ItoN}(\mathit{StVal}(l)), \mathit{ItoN}(l))\}$$
$$\text{where } G = (G_N, G_E)$$

The transfer function for a LOAD instruction is more complicated. Given a trace that ends with a LOAD instruction l, if the LOAD instruction loads from a memory location x, then the instruction prior to the LOAD instruction in the trace's slice is the STORE instruction that last stored to x. Thus, an edge is required from the store node representing x's MRU at the program point above l, to the load node representing l.

Before defining the transfer function for LOAD instructions, we will introduce a few concepts related to pointer analysis. *May* and *must* pointer information are important concepts in pointer analysis. In an analysis based on points-to sets, a pointer *may point to* a memory location if the address of the memory location is a possible value of the pointer. A pointer *must point to* a memory location if the address of the memory location is the only value of the pointer in all executions. In our analysis, we use a similar relation for may information: two nodes *value-alias* in an abstraction if the nodes may produce an address within a common object in some trace abstracted by the abstraction. This definition is field-insensitive.

Root nodes partition the memory space, meaning that there is a unique root node that represents all root instructions that may produce a particular address. If an execution trace ending with an instruction l produces an address with an object, then the slice of the trace is a sequence from the root instruction that produced the address, to l. Slices are represented as df-paths in a graph; thus, if two nodes are value-aliased, then there must be df-paths from a common root node to each of the two nodes.

Definition 3.10. Given an abstraction $G = (G_N, G_E) \in \mathit{Graph}$, the following notation is used to express direct and transitive df-paths between nodes:

$$n \rightarrow_G n' \equiv (n, n') \in G_E$$
$$n \rightarrow_G^+ n' \equiv (\exists \{n_1, \ldots, n_i\} \in \mathcal{P}(G_N) \mid n \rightarrow_G n_1 \wedge \ldots \wedge n_i \rightarrow_G n')$$

$n \rightarrow_G n'$ means that there is a edge between n and n'. $n \rightarrow_G^+ n'$ means that there is a df-path (possibly zero-length) from n to n'.

Suppose that a trace tr ends with a LOAD instruction represented by n and the execution of n at the end of the trace loads from a memory location within an object o. Then, n's d-source node must have produced an address within o. If there is a prefix tr' of tr that ends with a STORE instruction represented by n', and the execution of n' at the end of tr'

$$\mathit{ValAlias}, \mathit{DSrcd}, \mathit{FIReachDef} : \mathit{Graph} \times \mathit{Node} \to \mathcal{P}(\mathit{Node})$$

$$\mathit{ValAlias}(G, n) = \{n' \in G_N : (\exists n_r \in G_N \mid n_r \text{ is a root node} \wedge (n_r \to_G^+ n) \wedge (n_r \to_G^+ n'))\}$$

$$\begin{aligned}\mathit{DSrcd}(G, n) = \{n' \in G_N : \exists l, l' \in \mathit{ILabel} \\ \mid n' \text{ is a store node} \\ \wedge\, n = \mathit{ItoN}(l) \wedge n' = \mathit{ItoN}(l') \wedge l = \mathit{DSrc}(l')\} \\ \text{where } G = (G_N, G_E).\end{aligned}$$

$$\mathit{FIReachDef}(G, n) = \bigcup \{\mathit{DSrcd}(n') : n' \in \mathit{ValAlias}(G, n))\}$$

Figure 3.3: Utility functions

stores to o, then the d-source node of n' must have evaluated to an address within o as well. Thus, if a store node n' is a possible reaching definition to n, then the d-source of n and the d-source of n' must be value-aliased.

Definition 3.11. Several utility functions are defined in Figure 3.3:

- $\mathit{ValAlias} : \mathit{Graph} \times \mathit{Node} \to \mathcal{P}(\mathit{Node})$ is a function that maps an abstraction and a node n to all nodes that are value-aliased with n.
- $\mathit{DSrcd} : \mathit{Graph} \times \mathit{Node} \to \mathcal{P}(\mathit{Node})$ is a function that maps an abstraction and a node n to store nodes that are d-sourced from n.
- $\mathit{FIReachDef} : \mathit{Graph} \times \mathit{Node} \to \mathcal{P}(\mathit{Node})$ is a function that maps an abstraction and a node n to all store nodes that are d-sourced from a node that is value-aliased with n.

The flow-insensitive transfer function for LOAD instructions adds edges between a load node and all store nodes that may have stored to the object loaded by the load node:

$$\mathit{TsfrInstr}_N(G, [\![\mathit{LOAD}\ r\ f]\!]^l) = G_N \cup \{\mathit{ItoN}(l)\}$$
$$\mathit{TsfrInstr}_E(G, [\![\mathit{LOAD}\ r\ f]\!]^l) = G_E \cup \{(n', \mathit{ItoN}(l)) : n' \in \mathit{FIReachDef}(G, \mathit{DSrc}(l))\}$$
$$\text{where } G = (G_N, G_E)$$

3.3.2 Algorithm

A transfer function for an instruction l must only be applied to an abstraction G if G abstracts a trace that ends with an instruction immediately preceding l in the control flow graph. To ensure this, a set of instructions reached by an execution trace is recorded alongside the abstraction.

Definition 3.12. Given a set of instructions $R \in \mathcal{P}(ILabel)$, let $Next(R)$ be a set of instructions formed by instructions l such that there is a control flow edge from an instruction in R to l:

$$Next(R) = \bigcup \{l \in ILabel : (\exists l' \in R \mid (l', l) \in CFlow)\}$$

Given an abstraction G that abstracts a set of execution traces H, and a set of instructions R such that $\forall l \in R(\exists tr \in H \mid Last(tr) = l)$, let $Tsfr$ be a function such that $Tsfr(G, R) = (G', R \cup Next(R))$, where G' is an abstraction that abstracts all execution traces in H extended by a single instruction:

$$Tsfr(G, R) = \left(\bigsqcup \{TsfrInstr(G, l) : l \in Next(R)\}, R \cup Next(R)\right)$$

Then the abstraction that abstracts all execution traces can be computed by starting with the initial element $(\bot, \{l_{entry}\})$ and by iteratively applying $Tsfr$ until a fixed point is reached.

3.4 Overview of the Flow- and Field-sensitive Algorithm

The precision of an abstraction computed by the algorithm described in the previous section is less than desirable because the result is flow-insensitive and field-insensitive: it does not differentiate between different memory locations within an object, and it does not analyze whether a store *overwrites* other stores by being last to store to a memory location loaded by a LOAD instruction in all control flow paths to the LOAD instruction.

This section gives an overview of our flow-sensitive and field-sensitive algorithm. The algorithm computes an abstraction in a new abstraction domain that has nodes that do not represent instructions in a program.

D-offsets and modifiers are treated as integers up to the section on field-sensitivity.

The purpose or significance of concepts introduced in the first few subsections of this section may not be immediately apparent. A high-level overview of the subsections is given below:

- Consider the following segment of code:

```
1  ...
2  *p = ...;
3  ...
4  *p = ...;
5  ...
```

  Suppose that the flow-insensitive algorithm determined `2:L(*p)` and `4:L(*p)` to be possible reaching definitions of a load node. By manual analysis, we can conclude that `2:L(*p)` is not a reaching definition to the load node because `4:L(*p)` overwrites the value stored by `2:L(*p)` in all executions of this segment of code. We say that `4:L(*p)` *excludes* `2:L(*p)` from being a possible reaching definition of a load node. Reasoning whether one store overwrites another is simple with straight-line code, but becomes difficult with non-trivial control flow. Subsection 3.6.1 introduces a concept of inserting instructions into a trace that do not alter the state reached by executing the trace, but simplifies reasoning about complicated control flow.

- In the previous example, we assumed *a priori* that the two stores write to the same memory location. However, a statement on line 3 may have modified the variable `p`. Pointer information is required to safely assume that `p` is unmodified, because `p` may be modified by an indirect memory operation. Subsection 3.6.2 introduces a sub-analysis that determines whether there is a relation between values produced by executions of instructions.

- Consider the following segment of code:

```
1  ...
2  *p = ...;
3  ... = *p;
```

  Disregarding a degenerate case where `p` points to itself, we can conclude that `2:L(*p)` is last to store to the memory location loaded by `3:R(*p)`, and thus `2:L(*p)` is the only reaching definition of `3:R(*p)`. Subsection 3.6.3 combines concepts introduced in previous subsections to reason about possible reaching definitions of a specific load node.

3.4.1 High-level Algorithm

The full description of our flow-sensitive and field-sensitive algorithm is in Section 3.9. A high-level overview of the algorithm is given below:

1. Compute supporting data structures of the control flow graph.

2. While the abstraction has not reached a fixed point...

   (a) For each instruction l in a program, *process l*...

      i. Remove all nodes from the abstraction that are associated with l: due to loops in the control flow graph, an instruction may be processed more than once. When an instruction is reprocessed, the changes made to the abstraction when the instruction was previously processed are reverted.

      ii. Add nodes to the abstraction that are associated with l

      iii. For all nodes associated with l, compute their incoming edges (possible reaching definitions).

Determining the nodes added to an abstraction when a particular instruction is processed is not computationally intensive:

Given an instruction l...

1. If l is a LOAD(AM) instruction, then there is a unique load node associated with l that is added to the abstraction when l is processed.

2. If l is a STORE instruction, then l may be associated with different store nodes during the analysis, but l is never associated with more than one store node at a time.

3. If l is a SKIP instruction, then multiple nodes of a new type of node called ϕ *nodes* may be associated with l at the same time.

Computing the incoming edges of a node is where the most computation is performed in the algorithm. An overview of the steps involved is provided below:

Given a node n...

1. If n is a root node, do nothing.

2. If n is a store node, add an edge from the node representing the store-value instruction of the STORE instruction represented by n, to n.

3. If n is a node that loads from memory (load nodes and ϕ nodes)...

   (a) Find all nodes value-aliased with n's d-source node.

   (b) Determine the offsets within objects loaded by n (Section 3.8).

   (c) Determine the possible reaching definitions of n (Section 3.6).

   (d) Determine if unanalyzable constructs preclude the computation of a precise set of possible reaching definitions (Section 3.7).

3.5 Abstraction Domain $Graph_\phi$

This section introduces a new abstraction domain that incorporate a new type of node called ϕ nodes, which are nodes that simplify the computation of a set of possible reaching definitions.

3.5.1 Definitions

Definition 3.13. The abstraction domain $Graph_\phi$ is defined below.

$$\begin{aligned}
Graph_\phi &= \mathcal{P}(Node_\phi) \times \mathcal{P}(Edge_\phi) \\
Node_\phi &= Node_{\phi_D} \cup Node_{\phi_A} \cup Node_{\phi_M} \\
Node_{\phi_D} &= ILabel \times Node_\phi \times \mathbb{Z} \\
Node_{\phi_A} &= Var \\
Node_{\phi_M} &= ILabel \\
Edge_\phi &= Node_\phi \times Node_\phi
\end{aligned}$$

Many terms related to *Graph* have similar meanings in $Graph_\phi$. An *abstraction* in $Graph_\phi$ is a directed graph with nodes $Node_\phi$ and edges $Edge_\phi$. There are three categories of nodes. $Node_{\phi_D}$ is a set of *dereference nodes*. A dereference node n is referenced as a triplet $(l, n_s, f) \in Node_{\phi_D}$, where l is its position (defined below), n_s is its d-source node, and f is its d-offset. $Node_{\phi_A}$ is a set of *address nodes*, which are nodes that represent LOADA instructions. $Node_{\phi_M}$ is a set of *malloc nodes*, which are nodes that represent LOADM instructions. Store nodes are nodes that represent a STORE instruction. Load nodes are nodes that represent a LOAD instruction. Store and load nodes are dereference nodes. Address nodes and malloc nodes are *root nodes*.

Definition 3.14. Given a node $n \in Node_\phi$, let its *position* in the control flow graph of a program be the instruction $Pos_\phi(n)$, where Pos_ϕ is defined as follows:

Let $l, \in ILabel, n_s \in Node_\phi, f \in \mathbb{Z}$.

$$Pos_\phi : Node_\phi \to ILabel$$
$$Pos_\phi(n) = \begin{cases} l & \text{if } n = (l, l_s, f) \in Node_{\phi_D} \\ l & \text{if } n = l \in Node_{\phi_M} \\ l_{entry} & \text{if } n \in Node_{\phi_A} \end{cases}$$

There is at most one store node positioned at a STORE instruction, and at most one load node positioned at a LOAD instruction.

The position of a node is relevant in a concept of a *ϕ-annotated execution environment*, which is a hypothetical execution of an execution trace with inserted instructions that do not change the values stored in memory locations, but may affect the MRUs of memory locations.

Definition 3.15. Given an abstraction $G \in Graph$ and an execution trace tr abstracted by G, a *node-trace*, or *n-trace*, of tr is a sequence of nodes, constructed by inspecting the instructions in tr. For each instruction l in tr, the node positioned at l is added to the n-trace. If multiple nodes are positioned at l, then the nodes are sequenced in an arbitrary order[3]. An *n-trace abstracted by G* is an n-trace constructed from an execution trace that is abstracted by G.

[3]The order is irrelevant because the algorithm that computes an abstraction does not assume that multiple nodes that share a position are sequenced in a particular order

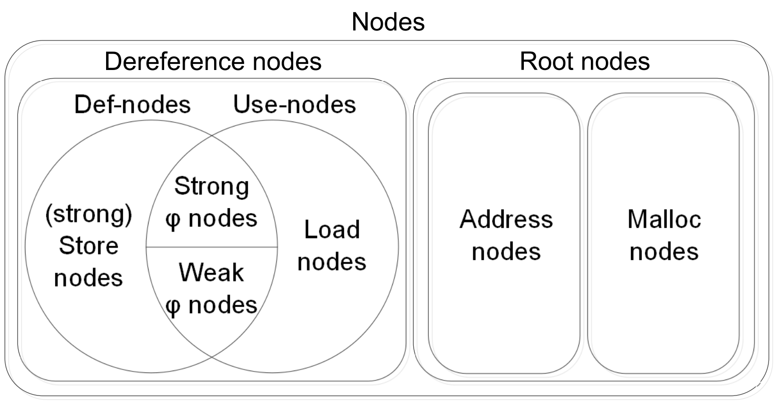


Figure 3.4: Node hierarchy

We give a semantic meaning to an *execution of a node*, which may store to a memory location, and reason about a program state reached by executing an n-trace. Nodes operate on a representation of a program state that, in addition to recording values stored in memory locations, it records the last node that stored to each memory location. The node that last stored to a memory location is called the *node-MRU*, or *nMRU*, of the memory location.

Definition 3.16. When executed, a def-node *produces* the value that it stores to memory, and a use-node produces the value that it loads from memory.

A ϕ *node* is a node that does not represent an instruction in a program. When a ϕ node is executed, it defines a memory location and changes its nMRU, but it does not change the value stored in the memory location. A ϕ node is a dereference node and has a d-source and a d-offset and its position is always a SKIP instruction. Unlike store and load nodes, multiple ϕ nodes may share the same position.

Store nodes are classified as a *def-node*. Load nodes are classified as a *use-node*. ϕ nodes are classified as both a def-node and a use-node. Def-nodes are further classified as either a *strong def-node* or a *weak def-node*: a *strong def-node* is dominated by its d-source node; a def-node is weak otherwise. A store node is always a strong def-node, and the semantics of a strong def-node is consistent with the semantics of a STORE instruction. Strong and weak def-nodes have different execution semantics. Figure 3.4 is a diagram of the hierarchy of nodes.

In the ϕ-annotated execution environment, we are only interested in changes to the nMRUs of memory locations. Thus, we specify that an execution of a def-node *defines* a memory location instead of specifying that it stores a value to a memory location.

Definition 3.17. The execution of a strong def-node $n = (l, n_s, f) \in Node_{\phi_D}$ at the end of an n-trace *tr* defines the memory location whose address is $x + f$ where x is equal to the value produced by the last execution of n_s in *tr*.

The d-source node of a strong def-node *n* dominates *n*; therefore the d-source node must have executed in an n-trace that reaches *n*.

Definition 3.18. The execution of a weak def-node $n = (l, n_s, f) \in Node_{\phi_D}$ at the end of an n-trace *tr* defines the memory locations whose address is $x + f$ if all of the following conditions are satisfied:

- n_s is executed in *tr*.
- The last execution of n_s in *tr* produces a value x.
- The d-source node of $x + f$'s nMRU is n_s.

3.5.2 ϕ Nodes

ϕ nodes are inserted below *join-points*, which are program points above instructions that have multiple predecessors in the control flow graph. The precise requirements that determine the placement of ϕ nodes and their reaching definitions are described in this subsection.

Definition 3.19. Given an instruction *l*, let $Df(l)$ be the dominance frontier of *l*, which is a set formed by all instructions l' such that *l* dominates one of l''s immediate predecessors in the control flow graph, but *l* does not strictly dominate l' [5]. Given an instruction *l*, let $Df^C(l)$ be *l*'s *dominance frontier closure*, which is the closure of the dominance frontier set with itself: $Df^C(l) = \bigcap\{S \in \mathcal{P}(ILabel) : S = Df(S) \wedge l \in S\}$.

In Chapter 2, we assumed that the input program has a SKIP instruction below every join-point. Thus, all instructions in a dominance frontier set are SKIP instructions. ϕ nodes are positioned at SKIP instructions below join-points.

An abstraction must satisfy the following property:

```
 1  while(...)
 2      skip; /* 2:P(a), 2:P(p), 2:P(*p) */
 3      a = &x;
 4      if(...) {
 5          p = &a;
 6          if(...) {
 7              *p = &y;
 8          }
 9          skip; /* 9 : P(*p) */
10      }
11      skip; /* 11 : P(p), P(*p) */
12  }
```

Figure 3.5: Example of def-node semantics and ϕ node placement

Definition 3.20. The *dominating def-node property* is as follows: given an abstraction $G \in Graph_\phi$, and a trace tr (not an n-trace) abstracted by G that executes a STORE instruction l represented by a store node $n = (l, n_s, f)$, for all instructions l' in the dominance frontier closure of l such that l' appears after l in tr, there is a ϕ node (l', n_s, f) in G. We say that n *induces* ϕ nodes at instructions in its dominance frontier closure.

ϕ nodes must preserve slice paths: if an abstraction $G \in Graph$ abstracts a set of execution traces and G has a flow edge from a store node n to a load node n', then an abstraction $G' \in Graph_\phi$ that abstracts the same set of execution traces must have a path from n to n' passing only through ϕ nodes. ϕ nodes are classified as both def-nodes and use-nodes because, conceptually, a ϕ nodes loads a value from a memory location and stores the value back to the same memory location. To preserve slice paths, a ϕ node's set of possible reaching definitions must include the nMRUs of memory locations defined by the ϕ node in every n-trace abstracted by G.

3.5.3 Example

A ϕ node is associated with a *virtual expression*: given a ϕ node $n = (l, n_s, f)$, n's virtual expression is `*(`$e + f$`)`, where e is the expression associated with n_s. In examples, a ϕ node (l, n_s, f), with a virtual expression e, is labelled k`:P(`e`)`, where k is the statement label of the C statement that immediately follows l in the control flow graph.

Figure 3.5 is a listing of an example C program. In this example only, "skip statements" are present to emphasize join-points.

Six ϕ nodes are necessary to satisfy Definition 3.20:

- `3:L(a)` induces `2:P(a)`;
- `5:L(p)` induces `2:P(p)` and `11:P(p)`;
- `7:L(*p)` induces `2:P(*p)`, `9:P(*p)`, and `11:P(*p)`.

`9:P(*p)` is a strong def-node because it is dominated by its d-source node, `5:L(p)`. `11:P(*p)` is not dominated by its d-source node, `5:L(p)`, and thus it is a weak def-node.

Consider the following n-trace:

$$tr = [\texttt{2:P(a)}, \texttt{2:P(p)}, \texttt{2:P(*p)}, \texttt{3:L(a)}, \texttt{5:L(p)}, \texttt{7:L(p)}, \texttt{7:L(*p)}, \texttt{9:P(*p)}]$$

The last execution of `5:L(p)` before the trace reached `9:P(*p)` produced the value $(\texttt{a}, 0)$, which is the address of the memory location at offset 0 in the object associated with `a`. Thus, `9:P(*p)` defines $(\texttt{a}, 0)$ without changing its value, and the nMRU of $(\texttt{a}, 0)$ at the end of *tr* is `9:P(*p)`.

Consider an extended trace:

$$tr' = tr \cdot [\texttt{11:P(p)}, \texttt{11:P(*p)}]$$

The execution of `11:P(*p)` at the end of tr' defines $(\texttt{a}, 0)$, because the last execution of its d-source node `5:L(p)` produced $(\texttt{a}, 0)$ and the nMRU of $(\texttt{a}, 0)$ is `9:L(*p)`, which has the same d-source as `11:P(*p)`.

Consider an extended trace:

$$tr'' = tr' \cdot [\texttt{2:P(a)}, \texttt{2:P(p)}, \texttt{2:P(*p)}, \texttt{3:L(a)}, \texttt{11:P(p)}, \texttt{11:P(*p)}]$$

The execution of `11:P(*p)` at the end of tr'' does not define any memory location because it is a weak def-node. Although the last execution (in the previous loop iteration) of its d-source node produced $(\texttt{a}, 0)$, the nMRU of $(\texttt{a}, 0)$ before executing `11:P(*p)` is `3:L(a)`. The d-source of `3:L(a)` is &a and the d-source of `11:P(*p)` is `5:L(p)`. Thus, `11:P(*p)` does not define $(\texttt{a}, 0)$ because the d-sources do not match, and the nMRU of $(\texttt{a}, 0)$ after the execution of tr'' is `3:L(a)`.

3.6 Flow-sensitivity

This section uses properties of abstractions in the abstraction domain $Graph_\phi$ to determine whether a def-node is a possible reaching definition of a use-node. The first three subsections define *precise* properties of the abstraction. These precise properties are statically uncomputable. The last subsection defines approximations to the precise properties that are used by our algorithm.

3.6.1 Def-groups

Def-groups are sets of def-nodes and have properties that simplify the computation of possible reaching definitions.

Definition 3.21. The set of *def-groups* of an abstraction is a partition of def-nodes such that def-nodes with the same d-source and d-offset are in the same def-group. The *d-source and d-offset of a def-group* are the d-source and d-offset that is common to all def-nodes in the def-group. Given an abstraction $G \in Graph_\phi$, the notation $Dg(G, n, f)$ is used to refer to a def-group with a d-source of n and a d-offset of f.

To enable the computation of a set of possible reaching definitions of ϕ nodes, we provide an algorithm that answers a slightly more general query than what is necessary to compute a set of possible reaching definitions of a load node.

Definition 3.22. Given an abstraction $G \in Graph_\phi$, a point t, a node n, and an offset f, let $ReachDef(G, t, n, f)$ be the set of nMRUs of memory locations whose address is $x + f$, where x is a value produced by the last execution of n before reaching t, in some program state reached by an n-trace that ends at t and is abstracted by G. Determining a superset of $ReachDef(G, t, n, f)$ is called a *reaching definition query*.

ReachDef is a precise set of possible reaching definitions. However, *ReachDef* is uncomputable because computing it requires enumerating all execution traces abstracted by an abstraction. Given G, t, n, and f as defined above, our algorithm computes a superset of $ReachDef(G, t, n, f)$. Our objective is to be as precise as possible, which means that we want our approximation to be as close to $ReachDef(G, t, n, f)$ as possible. A node is *definitely not* a reaching definition if it can be shown not to be in the precise set.

Given a load node $n = (l, n_s, f)$ and an abstraction $G \in Graph_\phi$, a superset of $ReachDef(G, PtAbove(l), n_s, f)$ is the most precise set of possible reaching definitions of n. We want to compute an approximation of *ReachDef* at non-join-points, because the dominating def-node property (Definition 3.20) can be used to determine that certain nodes are definitely not reaching definitions. The points immediately prior to join-points in a control flow graph are not join-points (non-join-points) because we assume that the control flow graph consists of elementary blocks; thus a conservative set of possible reaching definitions of a ϕ node $n = (l, n_s, f)$ can be computed by computing sets of possible reaching definitions at each of the immediate predecessors of $PtAbove(l)$, and then taking the union of the sets.

Given an abstraction $G \in Graph_\phi$, suppose that the algorithm is processing a LOAD instruction and that it needs to compute a conservative set of possible reaching definitions for the load node $n = (l, n_s, f)$ that represents the instruction. First, $ValAlias(G, n_s)$ is computed, which is the set of nodes value-aliased with n's d-source node n_s. Only def-nodes that are d-sourced from a node in $ValAlias(G, n_s)$ may store to the same object loaded by n. If the set of def-nodes that are d-sourced from a node in $ValAlias(G, n_s)$ is partitioned by common d-source and d-offset into def-groups, the following proposition can be used to show that, under certain conditions, only one def-node in each def-group can be a reaching definition to the load node.

Definition 3.23. Given a set of nodes N and a point t, if there exists a node n in N such that, (**i**) n dominates t, and (**ii**) no node in N dominates t and is strictly dominated by n, then n is called the *immediate node* of N at t.

Proposition 3.24. Given an abstraction $G \in Graph$, a non-join-point t, a def-group $N = Dg(G, n, f)$, and an n-trace tr that is abstracted by G and ends at t, if the last execution of n in tr produced a value x, then the nMRU of $x + f$ at the end of the trace either is the immediate node of N at t, or is not a node in N.

Proof. First, a trivial case is handled: if tr did not execute a def-node in N, then the nMRU of $x + f$ is not a node in N.

Let tr execute a def-node in N. By the dominating def-node property, N has an immediate node at t. Let n_i be N's immediate node at t. Let the last execution of n in tr produce an arbitrary value x. Let $\hat{n}$, $\hat{n} \neq n_i$, be a def-node in N. We will show that, in tr, if $\hat{n}$ defines $x + f$, then n_i defines $x + f$ after $\hat{n}$ defines $x + f$; therefore, $\hat{n}$ is not the nMRU of $x + f$ at t.

Let an execution of $\hat{n}$ in tr define $x + f$. The following property of immediate nodes is useful in proving that $\hat{n}$ is not the nMRU of $x + f$. All cf-paths from $\hat{n}$ to t must contain n_i:

- If $\hat{n}$ dominates t, then $\hat{n}$ must strictly dominate n_i because n_i is the immediate node of the def-group.
- If $\hat{n}$ does not dominate t, then by the dominating def-node property, tr must execute a ϕ node n' in N such that n' dominates t. Then, either $n' = n_i$ or n' strictly dominates n_i because n_i is the immediate node of N.

There are two cases to consider, depending on whether n executes again after $\hat{n}$ defines $x + f$:

- Suppose that n is not executed again after $\hat{n}$ defines $x + f$. If the nMRU of $x + f$ when the n-trace reaches n_i is $\hat{n}$, then n_i defines $x + f$ after $\hat{n}$ defines $x + f$ because all paths from $\hat{n}$ to t must contain n_i[4].
- Suppose that n is executed again after $\hat{n}$ defines $x + f$. Then, there exists a cf-path from $\hat{n}$ to n. All cf-paths from $\hat{n}$ to t must contain n_i. There are two cases to consider: either all cf-paths from $\hat{n}$ to n contain n_i or all cf-paths from n to t contain n_i.
  - Suppose that all cf-paths from $\hat{n}$ to n contain n_i. Then n_i defines $x + f$ after $\hat{n}$ defines $x + f$ because n produced the value $x + f$ for $\hat{n}$ to have defined $x + f$, and n_i is executed before n can produce a different value.
  - Suppose that all cf-paths from n to t contain n_i. Then n_i defines $x + f$ after $\hat{n}$ defines $x + f$ because n produces the value $x + f$ in its last execution before reaching t, and there are no cf-paths from n_i to t that contain $\hat{n}$ but not n_i.

$\square$

[4]The nMRU is relevant because n_i may be a weak def-node. Weak def-nodes may or may not define a memory location depending on the nMRU of the memory location.

Definition 3.25. The *residual def-node* of a def-group N is the immediate node of N at the point above the position of the d-source node of N, if such an immediate node exists.

The next proposition is an important result in this thesis. The proposition states that, in a reaching definition query, all but two def-nodes in a def-group are definitely not reaching definitions.

Proposition 3.26. Given an abstraction $G \in Graph_{\phi}$, a non-join-point t, a node n, an offset f, and a def-group $N = Dg(G, n', f')$, the immediate node of N at t and the residual def-node of N are the only nodes that may be in both N and $ReachDef(G, t, n, f)$.

Proof. Trivial cases are handled first:

- If n is not executed in tr, then $ReachDef(G, t, n, f) = \emptyset$.
- If G does not abstract an n-trace that executes a def-node in N and ends at t, then $N \cap ReachDef(G, t, n, f) = \emptyset$.
- If G does not abstract an n-trace where an execution of n produces an arbitrary address x and an execution of n' produces $x + f - f'$, then no def-node in N defines $x + f$, and thus $N \cap ReachDef(G, t, n, f) = \emptyset$.

Let G abstract an n-trace tr that executes a def-node in N and ends at t. Let n_i be N's immediate node at t (which must exist if a def-node in N is executed in tr). Let the last execution of n in tr produce an arbitrary address x. Let n' produce $x + f - f'$ in some execution in tr. We will show that the nMRU of $x + f$ either is n_i, is the residual node of N, or is not a def-node in N.

There are two cases to consider, depending on whether the last execution of n' in tr produces $x + f - f'$:

- Suppose the last execution of n' in tr produced $x + f - f'$. Then by Proposition 3.24, the nMRU of $x + f$ either is n_i or is not a def-node in N.
- Suppose the last execution of n' in tr did not produce $x + f - f'$. Let t' be the point above the position of n', and let tr' be the longest prefix of tr such that it ends at t' and the last execution of n' in tr' produces $x + f - f'$. By Proposition 3.24, in the

program state reached by tr', the nMRU of $x + f$ either is the immediate node of N at t' i.e. the residual def-node of N, or is not a def-node in N. Between the end of tr' and tr, n' does not produce $x + f - f'$. Therefore, an execution of a def-node in N does not define $x + f$ between the end of tr' and tr.

Therefore, the immediate node of N at t and the residual def-node of N are the only nodes that may be in both N and $ReachDef(G, t, n, f)$.

□

An important performance consideration is how the immediate def-node of a def-group at a point can be determined. Chase *et al.*'s paper describes a data structure called a *skeleton tree*: a skeleton tree is a subgraph of a dominator tree, induced by a subset of the nodes that contain a definition to a particular variable [2]. Finding the reaching definition of a variable at a point can be performed in $O(\log(n))$ time, where n is the number of nodes in the skeleton tree associated with the variable [7]. Performance-wise, a simpler data structure may be adequate: an *assignment list* is a list of definitions to a particular variable, and the list is ordered such that no definition is preceded by a definition that dominates it [29]. Finding a reaching definition of a variable at a point is performed by finding the first definition in the list that dominates the point. We use an assignment list to index def-nodes in a def-group.

3.6.2 Store Node Promotion

In the previous section, we have showed that the number of def-nodes in a particular def-group that is also in a *ReachDef* set is at most two. However, each store instruction has a unique d-source instruction, and each instruction except for LOADA instructions is represented by a unique node. Therefore, a node is a d-source of more than one store node only if it is an address node. There is no precision benefit to forming def-groups that contain only one store node.

A concept that reduces the number of non-empty def-groups is that we may replace a store node with a *promoted* store node that has a different d-source and d-offset if the promoted store node has exactly the same execution behaviour as the original store node in all n-traces abstracted by an abstraction. The semantics of a def-node allows this transformation. Consider a store node $n = (l, n_s, f)$. The d-source of a store node always dominates the store node, which is a property that will be preserved. Given an abstraction $G \in Graph_\phi$, suppose that there exists a constant integer $\bar{f}$ and a node n'_s that dominates n_s such that in all n-traces that are abstracted by G, if n_s produces a value x in some execution, then n'_s produces a value $x - \bar{f}$ in its last execution before the execution of n_s. Then a promoted store node $(l, n'_s, \bar{f} + f)$ has the same execution behaviour as n in all n-traces that are abstracted by G.

Definition 3.27. Given an abstraction $G \in Graph_\phi$ and a node n, the *equal set* of n, denoted $EqualSet(G, n)$, is formed by the nodes n' such that n' dominates n, and there exists a constant $\bar{f}$ such that in all n-traces that are abstracted by G, if n produces a value x in some execution, then n' produces a value $x - \bar{f}$ in its last execution before the execution of n. $\bar{f}$ is called the *constant relative offset* of n' relative to n.

It is obvious that if n' is in the equal set of n and n'' is in the equal set of n', then n'' is in the equal set of n. Thus, to replace store nodes with promoted store nodes that minimize the number of non-empty def-groups, a store node (l, n_s, f) is replaced by a promoted store node (l, n'_s, f) where n'_s is the node in $EqualSet(G, n_s)$ that dominates all other nodes in $EqualSet(G, n_s)$.

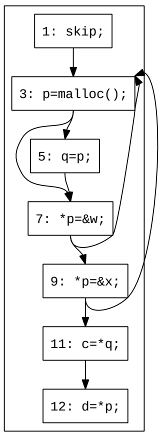
The graph in Figure 3.6(c) is an abstraction of Figure 3.6(a). The notation &3 in Figure 3.6(c) corresponds to a malloc node that represents the dynamic allocation function in line 3 of Figure 3.6(a). `7:L(*p)` and `9:L(*p)` are promoted store nodes. The d-source of the store nodes that were replaced by `7:L(*p)` and `9:L(*p)` is `7:L(p)` and `9:L(p)`, respectively. The equal set of `7:L(p)` and `9:L(p)` is $EqualSet(G,$ `7:L(p)`$)$ = {`7:L(p)`, `3:L(p)`, &3} and $EqualSet(G,$ `9:L(p)`$)$ = {`9:L(p)`, `3:L(p)`, &3}, respectively. &3 is closer to the root of the program's dominator tree than `7:L(p)` and `9:L(p)`. Therefore, the promoted store nodes have a d-source of &3.

3.6.3 Must Definitions

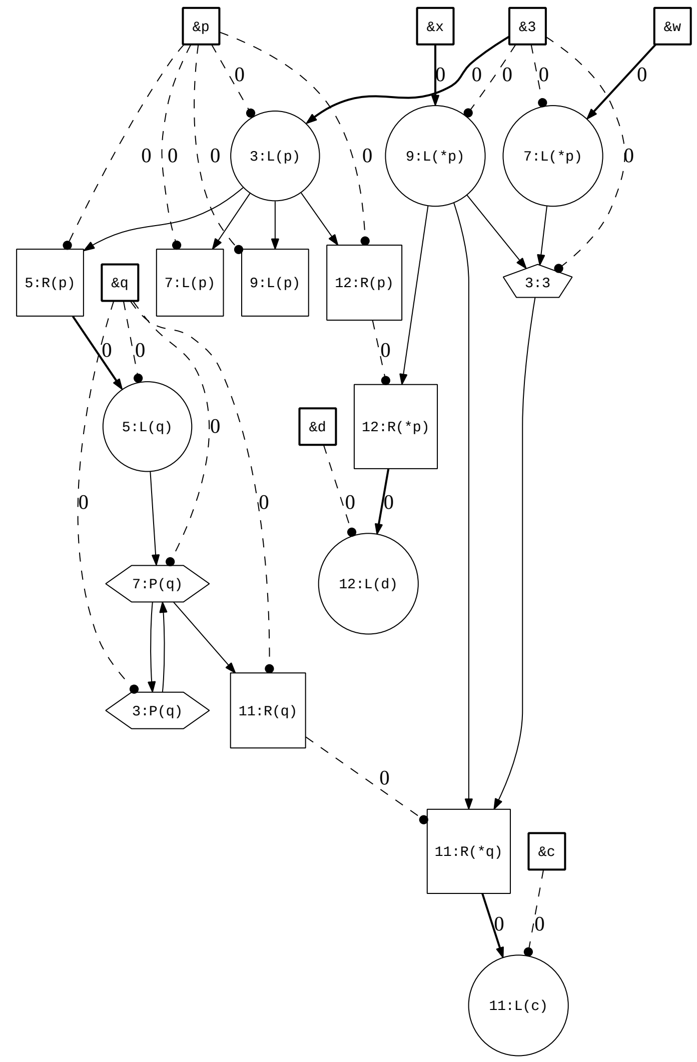
Immediate and residual def-nodes of a def-group exclude other def-nodes in the def-group from being in a *ReachDef* set. In this section, we describe how a def-node in one def-group can exclude def-nodes in another def-group.

```
1   while(...) {
2       while(...) {
3           p = malloc();
4           if(...) {
5               q = p;
6           }
7           *p = &w;
8       }
9       *p = &x;
10  }
11  c = *q;
12  d = *p;
```

(a) Code



(b) Control flow graph



(c) Graph

Figure 3.6: Example of store node promotion

Definition 3.28. Given an abstraction $G \in \mathit{Graph}$, a non-join-point t, a node n and an offset f, a def-node $n' = (l', n'_s, f') \in \mathit{Node}_{\phi_D}$ is a *must definition* at t if the following conditions are all true:

- n' dominates t.
- n'_s dominates n' (n' is a strong def-node).
- n'_s is in the equal set of n.
- The constant relative offset of n'_s relative to n is equal to $f' - f$.

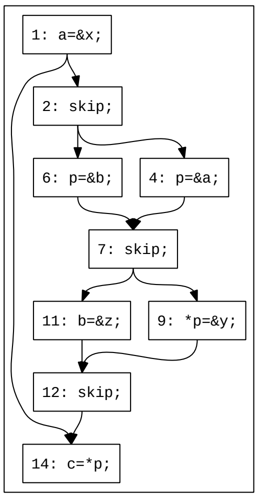
Proposition 3.29. Given an abstraction G, a non-join-point t, a node n, and an offset f, if a def-node n' is a must definition at t, then a def-node $\hat{n}$ that strictly dominates n' is not in $\mathit{ReachDef}(G, t, n, f)$.

Proof. Let $n' = (l', n'_s, f')$ be a must definition at t. Suppose that a def-node $\hat{n}$ strictly dominates n'. All cf-paths from $\hat{n}$ to t must pass through n' because n' dominates t. Let tr be an arbitrary n-trace that is abstracted by G and ends at t. tr does not execute n'_s after the last execution of n' in tr, because n'_s dominates n' and n' dominates t. If the last execution of n in tr produces an arbitrary address x, then the last execution of n'_s produces $x - f' + f$, because n'_s is in the equal set of n and the constant relative offset of n'_s relative to n is $f' - f$. Therefore, n' must define $x + f$ after $\hat{n}$. □

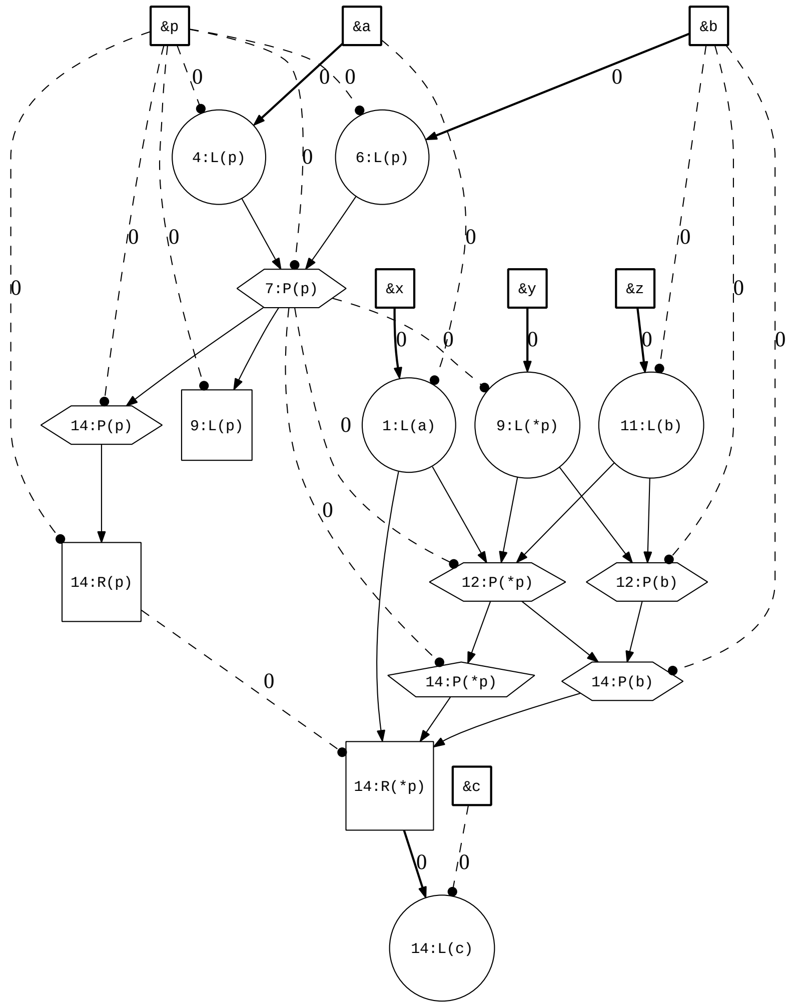
The graph in Figure 3.7(c) is an abstraction of Figure 3.7(a). Strong ϕ nodes are hexagons in the figure and they are `7:P(p)`, `12:P(b)`, `12:P(*p)`, `14:P(b)`, and `14:P(p)`. `12:P(*p)` has possible reaching definitions from three def-groups; the def-groups have a d-source node of `7:P(p)`, &a, and &b. `14:P(*p)` is a weak ϕ node because it's d-source node, `7:P(p)`, does not dominate it. It's reaching definition comes only from the def-group with a d-source of `7:P(p)`. Although `7:P(p)` is in the equal set of `14:R(p)`, `14:P(*p)` does not meet the requirements of a must definition because it is a weak def-node, and thus, `1:L(a)` is a possible reaching definition to `14:R(*p)`.

```
1   a = &x;
2   if(...) {
3       if(...) {
4           p = &a;
5       } else {
6           p = &b;
7       }
8       if(...) {
9           *p = &y;
10      } else {
11          b = &z;
12      }
13  }
14  c = *p;
```

(a) Code



(b) Control flow graph



(c) Graph

Figure 3.7: Example of strong and weak ϕ nodes

3.6.4 Approximations

The previous sections listed precise definitions for the set of ϕ nodes required to satisfy the dominating def-node property, the equal set, and the possible reaching definition set. The precise definitions require enumerating all traces abstracted by an abstraction; thus the precise sets are uncomputable in a static analysis. This subsection describes the approximations computed by our algorithm. All modifiers of store nodes are assumed to be 0. The changes required to handle non-zero modifiers are described in the section on field-sensitivity (Section 3.8).

Nodes

The definition of an equal set in Definition 3.27 expresses a precise set. A simple algorithm finds a conservative approximation of an equal set: to compute the equal set of n, trace the longest backward path starting from n; in a backward trace, if there is an edge $(i, j) \in G_E$, node j is visited before node i. Node i is part of the longest path if i dominates j and j has only one incoming df-edge (from i). Weak def-nodes are not placed in the conservative approximation of an equal set, because such nodes are not guaranteed to produce a value in every execution.

The algorithm uses the following reasoning: By definition, we know that a node n is in the equal set of itself. Given an abstraction G and a node n, suppose that there exist nodes n' and n'' such that n'' dominates n', n' dominates n, n' is in the equal set of n, and n' has a single incoming edge from n''. For all execution traces tr that are abstracted by G and end with n, consider the last execution of n' in tr. n' is in the equal set of n, thus the execution of n' produces a value that differs by a constant offset from the value produced by n at the end of tr. The sole edge from n'' to n' indicates that n' produces a value that differs by a constant offset from the value produced by the last execution of n'' in tr. All control flow paths from n'' to n passes through n'. Thus n'' is in the equal set of n.

Given an abstraction $G \in Graph_\phi$, let the approximate equal set for a node n, computed using the algorithm above, be denoted $AEqualSet(G, n)$. Given a set of nodes S that is totally ordered by the dominator relation, let $MaxDom(S)$ be the node that dominates all nodes in S. $EqualSet(G, n)$ and $AEqualSet(G, n)$ are totally ordered by the dominator relation. Let

$Promote(G, n) = MaxDom(AEqualSet(G, n)))$. $Promote(G, n)$ determines the d-source of a promoted store node.

The function that determines the nodes added to an abstraction when an instruction is processed can now be defined.

Definition 3.30. Given an abstraction $G \in Graph_\phi$ and a LOAD(AM) or STORE instruction l, let $ItoN(G, l)$ be l's representative node:

$$ItoN : Graph_\phi \times ILabel \to Node_\phi$$

$$ItoN(G, l) = \begin{cases} v & \text{if } l \text{ labels a “LOADA } r\ v\text{” instruction,} \\ & \quad \text{where } r \in R, v \in Var \\ l & \text{if } l \text{ labels a LOADM instruction} \\ (l, ItoN(DSrc(l)), DOff(l)) & \text{if } l \text{ labels a LOAD instruction} \\ (l, Promote(G, ItoN(DSrc(l))), & \\ \qquad DOff(l)) & \text{if } l \text{ labels a STORE instruction} \end{cases}$$

Given an instruction l, $ItoN$ maps l to a node that represents l. LOAD, LOADM, and STORE instructions are each represented by an unique node. LOADA instructions for a particular variable are mapped to a node unique to that variable.

Definition 3.20 dictates the placement of ϕ nodes in an abstraction. When an instruction l is processed, if l is in the dominance frontier closure of a STORE instruction represented by a store node (l', n'_s, f'), then a ϕ node (l, n'_s, f') is created to satisfy the dominating def-node property.

Definition 3.31. Given an abstraction $G \in Graph_\phi$ and a SKIP instruction l, let $ItoN_\phi(G, l)$

be the set of ϕ nodes required at l to satisfy the dominating def-node property:

$$ItoN_{\phi} : Graph_{\phi} \times ILabel \rightarrow \mathcal{P}(Node_{\phi})$$

$$\begin{aligned}
ItoN_{\phi}(G,l) = \{(l,n_s,f) : n_s \in G_N, f \in \mathbb{Z},\\
&(\exists l' \in ILabel\\
&\quad \mid l \in Df^C(l')\\
&\quad \wedge l' \text{ labels a STORE instruction}\\
&\quad \wedge ItoN(G,l') = (l',n_s,f))\}
\end{aligned}$$

Df-edges

Let *AReachDef* be an approximation of *ReachDef* that is computed by our algorithm. If $n = (l,n_s,f) \in Node_{\phi D}$ is a load node, then its set of possible reaching definitions is $AReachDef(G, PtAbove(l), n_s, f)$. If n is a ϕ node, then its set of possible reaching definitions is the union of the sets $AReachDef(G,t',n_s,f)$, where t' is a predecessor of $PtAbove(l)$ in the control flow graph.

Given an abstraction $G \in Graph_{\phi}$, a point t, a node n_s, and an offset f, the first step to computing $AReachDef(G,t,n_s,f)$ is to compute $ValAlias(G,n_s)$, which is the set of all nodes that are value-aliased with n_s. $ValAlias(G,n_s)$ is computed by traversing df-edges in the backward direction starting from n_s, until all root nodes that are value-aliased with n_s are found. Then df-edges are traversed in the forward direction starting from all value-aliased root nodes, until all nodes that are value-aliased with n_s are found. The running time of this step is worst-case linear in the number of edges between value-aliased nodes of n_s.

The set of def-nodes that are d-sourced from a node in $ValAlias(G,n_s)$ is a superset of $ReachDef(G,t,n_s,f)$. Proposition 3.24 states that only the immediate and residual def-nodes of def-groups are in $ReachDef(G,t,n_s,f)$. Thus, for each node $n'_s \in ValAlias(G,n_s)$, if $Dg(G,n'_s,f)$ has an immediate or residual def-node at t, then they are added to an intermediate set S of possible reaching definitions. S is a superset of $ReachDef(G,t,n_s,f)$, but its size can be further reduced by using must definitions. The running time to find the immediate node of a def-group is worst-case linear in the number of nodes in the def-group.

```
1  p = malloc();
2  *p = ...;
3  *p = ...;
4  ... = *p;
```
(a) Straight-line code

```
1  p = malloc();
2  *p = ...;
3  if(...) *p = ...;
4  else *p = ...;
5  skip;
6  ... = *p;
```
(b) Branching code

Figure 3.8: Simple example of flow-sensitivity

To find must definitions, the set $AEqualSet(G, n_s)$ is computed. The running time of the computation of $AEqualSet(G, n_s)$ is linear in the size of $AEqualSet(G, n_s)$. For each node $n'_s \in AEqualSet(G, n_s)$, if $Dg(G, n'_s, f)$ has an immediate node at t, then the immediate node of $Dg(G, n'_s, f)$ is added to a set M of must definitions. Let the immediate node of M be the *immediate must definition*; a def-node that strictly dominates an immediate must definition is definitely not a reaching definition. The def-nodes in S that do not strictly dominate the immediate node of M form the set $AReachDef(G, t, n_s, f)$.

3.6.5 Precision

The improvement in precision over the flow-insensitive algorithm, and also over points-to-set-based algorithms, is a consequence of the interaction between the dominating def-node property, def-group formation, store node promotion, and must definitions. To summarize how all the ideas come together to form a precise pointer analysis, code listings that are similar to the ones in the flow-sensitivity overview section (Section 3.4), are analyzed.

Consider the code listing in Figure 3.8(a). Given an abstraction G (computed by the flow-sensitive algorithm) that abstracts all traces that end at or before program point $\overline{4}$, suppose that the flow-sensitive algorithm is computing the possible reaching definitions of `4:R(*p)`. The nodes that are value-aliased with `4:R(p)` are &1 (the node that represents the LOADM instruction), `1:L(p)`, `2:L(p)`, `3:L(p)`, and `4:R(p)`. The load nodes `2:L(p)` and `3:L(p)` have &1 in their approximate equal sets. Thus, the d-source of `2:L(*p)` and `3:L(*p)` is &1, and `2:L(*p)` and `3:L(*p)` are in the same def-group, $Dg(\&1, 0)$. The immediate node of the def-group at $\overline{4}$ is `3:L(*p)`. Thus, the algorithm determines that `3:L(*p)` is the only reaching definition of `4:R(*p)`.

In the above example, the determination of the precise set of possible reaching definitions for 4:R(*p) (precise meaning the algorithm computes the same set as *ReachDef*) is trivial. If the code has control flow branches, the precise set of possible reaching definitions for a load node is not so apparent.

Consider the code listing in Figure 3.8(b). Suppose that the flow-sensitive algorithm is computing the possible reaching definitions of 6:R(*p). 2:L(*p) definitely stores to the memory location that is loaded by 6:R(*p). 3:L(*p) and 4:L(*p) may store to the memory location that is loaded by 6:R(*p). Given the information above, it may seem inevitable that an algorithm must determine that all three stores to *p are possible reaching definitions of 6:R(*p).

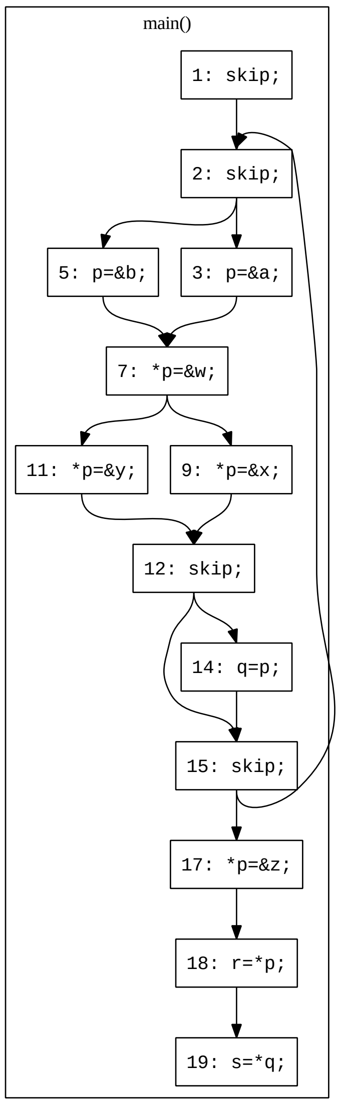
However, the dominating def-node property states that there must be a ϕ node 5:P(*p) in the abstraction. Due to store node promotion, the d-source node of 2:L(*p), 3:L(*p), 4:L(*p), and 5:P(*p) is &1. The immediate node of $Dg(\&1, 0)$ is 5:P(*p), and 5:P(*p) is the only reaching definition of 6:R(*p).

The mechanism of how the placement of ϕ nodes improves the precision of the computed approximation may not be obvious because ϕ nodes do not represent actual store instructions, and thus they preserve slice paths between actual stores and loads. The precision improvement is realized by how the algorithm computes the possible reaching definitions of 5:P(*p): the algorithm takes the union of the sets of possible reaching definitions computed at $\underline{3}$ and $\underline{4}$. 3:L(*p) is the immediate node of $Dg(\&1, 0)$ at $\underline{3}$, and thus 3:L(*p) is the only reaching definition of $AReachDef(G, \underline{3}, \&1, 0)$. Similarly, 4:L(*p) is the only reaching definition of $AReachDef(G, \underline{4}, \&1, 0)$. Thus, the algorithm determines that the possible reaching definitions of 5:P(*p) are 3:L(*p) and 4:L(*p). The presence of ϕ nodes enables the flow-sensitive algorithm to obtain precise results by utilizing "straight-line-code reasoning" in a structured way inside complex control flow graphs.

In a points-to-set-based analysis, an indirect strong update cannot be performed through the pointer "p" because "p" points to a non-singular abstract object: if the code listings are embedded in a loop, the abstract object pointed-to by "p" is responsible for abstracting the values of all dynamic objects allocated by calls to the dynamic allocation function. Thus the stores to "*p" are all *weak updates*: the resulting precision is equivalent to an EDF graph where all stores to *p are reaching definitions to the load of "*p".

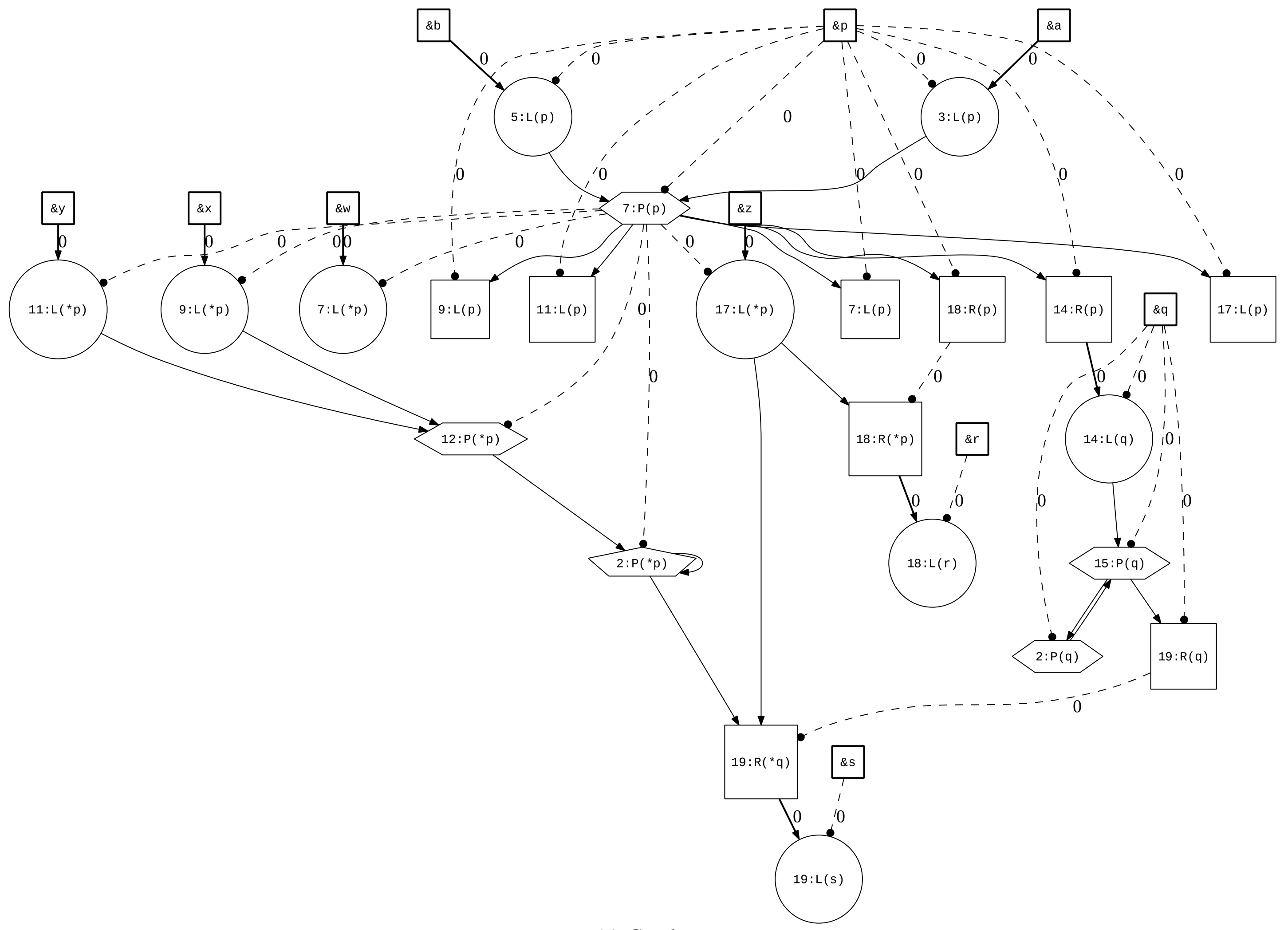
```
 1  while(...) {
 2      if(...) {
 3          p = &a;
 4      } else {
 5          p = &b;
 6      }
 7      *p = &w;
 8      if(...) {
 9          *p = &x;
10      } else {
11          *p = &y;
12      }
13      if(...) {
14          q = p;
15      }
16  }
17  *p = &z;
18  r = *p;
19  s = *q;
```

(a) Code



(b) Control flow graph

Figure 3.9: Example of flow-sensitivity



(c) Graph

Figure 3.9: Example of flow-sensitivity (cont.)

Figure 3.9 is another example that demonstrates the precision of the flow-sensitive algorithm in the presence of loops. Consider the load nodes 18:R(*p) and 19:R(*q). Their d-source nodes are 18:R(p) and 19:R(q), respectively. The def-nodes that are d-sourced from a node that is value-aliased with 18:R(p) or 19:R(q), which are the def-nodes that may potentially store to a memory location loaded by the load nodes, are 7:L(*p), 9:L(*p), 11:L(*p), and 17:L(*p). The four def-nodes are in the same def-group, $Dg(7\texttt{:P(p)}, 0)$. The immediate node of $Dg(7\texttt{:P(p)}, 0)$ at $\overline{18}$ and $\overline{19}$ is 17:L(*p). The residual of the def-group is 2:P(*p). The immediate node and the residual node are the possible reaching definitions of 19:R(*q). For 18:R(*p), 17:L(*p) is a must definition; 2:P(*p) strictly dominates 17:L(*p) and thus, the algorithm excludes the residual of the def-group, 2:P(*p), from being a possible reaching definition. Therefore, 17:L(*p) is the only reaching definition of 18:R(*p).

In a points-to-set-based analysis, all stores to "*p" are weak updates because the pointer "p" has two possible targets, "a" and "b", and strongly updating the abstract objects of either "a" or "b" is unsound. To determine the possible values of "*p" and "*q" on lines 18 and 19, a points-to-set-based analysis takes the union of the sets of possible points-to targets of "a" and "b". Thus a points-to-set-based analysis determines that "r" and "s" may point to "w", "x", "y", or "z". Our algorithm determines that "r" must point to "z" (&z is the only root node that is value-aliased with 18:L(r)), and that "s" may point to "x", "y", or "z".

3.7 Unanalyzable Expressions and Statements

This section covers the representation and manipulation of pointers that the analysis treats as possibly pointing to nearly all memory locations. There are many situations where such pointers may arise: for example, the right-hand side of an assignment may be an unanalyzable expression, or there may be call to a procedure that the analysis cannot analyze.

3.7.1 Topped Pointers

In an abstract interpretation of a program, the *top*, or $\top$, lattice element is an abstraction that describes any program behaviour. It is the most conservative interpretation of the program that is always sound. In a points-to abstraction, this is an abstraction that assumes that all memory locations contain any possible value. When an analysis determines that an individual pointer may have any value, we refer to the pointer as having *topped*. In our abstraction, the analogue to topped pointers are topped nodes.

The set of possible values of topped pointers can be refined. A well used concept in compiler optimizations is the static separation of *address-taken* and *non-address-taken* memory locations. The distinction is made at the object level: if any memory location inside an object has its address taken with an address-of operator, then all memory locations within the object is address-taken. All heap objects and all objects with external linkage are address-taken. An important feature of this classification is that non-address-taken objects can only be directly referenced through a variable.

A *store-all* is a store that may potentially store to any address-taken memory location. An indirect store through a topped pointer or an unanalyzable statement is a store-all. A simple method of handling store-alls in our abstraction would be to create a special "top" node and add a df-edge from all root nodes representing address-taken objects to the top node. A store-all can be represented as a store node that is d-sourced from the top node, which would make the store node a possible reaching definition to any load node that may potentially load from an address-taken memory location.

A *load-all* is a load that may potentially load from any address-taken memory location. An indirect load through a topped pointer is a load-all. Handling a load-all with a load node d-sourced from the "top" node is a poor design because a large portion of the entire graph is value-aliased with the top node, and all value-aliased nodes must be traversed when computing the possible reaching definitions of the load node. A more efficient method of representation is possible by over-approximating the effects of store-alls and load-alls.

3.7.2 Manipulations of Topped Pointers

Definition 3.32. A *topped root node* is a root node that may produce any value. A node *n* is *topped* if there exists a df-path from a topped root node to *n*.

Topped root nodes break the property that the sets of addresses produced by each root node partition the memory space. Thus, the algorithm that computes a set of possible reaching definitions must specially handle nodes that are df-reachable from a topped root node.

Definition 3.33. An *address-taken root node* is a root node that may produce the address of an address-taken memory location. A node *n* is *address-taken* if there exists a df-path from an address-taken root node to *n*.

If an address node has an outgoing df-edge, it is address-taken. Root nodes that represent LOADM instructions are address-taken. Thus the only nodes that are not address-taken are address nodes of non-address-taken variables (they do not have outgoing df-edges), and degenerate nodes that produce no values (they are not df-reachable from any root node).

A def-node is a *store-all* if it's d-source node is topped. A def-node is a *store-to-address-taken* if it's d-source node is address-taken or topped.

A store-all node is potentially a possible reaching definition to a large number of use-nodes. A load-all node potentially has a large number of possible reaching definitions. If a program has many store-all and load-all nodes, the graph can become quite dense.

A use-node is *use-topped* if our algorithm does not attempt to compute its set of possible reaching definitions, and instead marks it as a topped root node. By marking a use-node as use-topped if a store-all is a possible reaching definition, we avoid having nodes with high incoming and outgoing degrees. If information regarding the memory dependencies of store-all and load-all nodes is desired, then the information can be recovered post-analysis.

During the analysis, by over-approximating store-alls and load-alls, specific memory dependencies of store-all and load-all nodes are not required to perform a conservative analysis.

There are two cases where a use node is use-topped. These two cases are not mutually exclusive.

- If a use-node's d-source node is address-taken, then the use node is use-topped if a store-all node is a possible reaching definition to the use node. Store-all nodes are possible reaching definitions to all use nodes that are dereferenced from address-taken nodes; thus, unless the effects of a store-all have been overwritten by a must definition, the store-all is considered a possible reaching definition and the use node is use-topped as an over-approximation.

- If a use node's d-source node is topped, then the use node is use-topped if a store-to-address-taken node is a possible reaching definition to the use node. Store-to-address-taken nodes are possible reaching definitions to all load-all nodes; thus, unless the effects of a store-to-address-taken have been overwritten by a must definition, the store-to-address-taken node is considered a possible reaching definition and the use node is use-topped as an over-approximation.

To support unanalyzable constructs, the representative instruction set is extended with two new instruction types:

- Let "LOADT r" be an instruction that writes some indeterminable address of an address-taken memory location to the register r. LOADT is a root instruction.

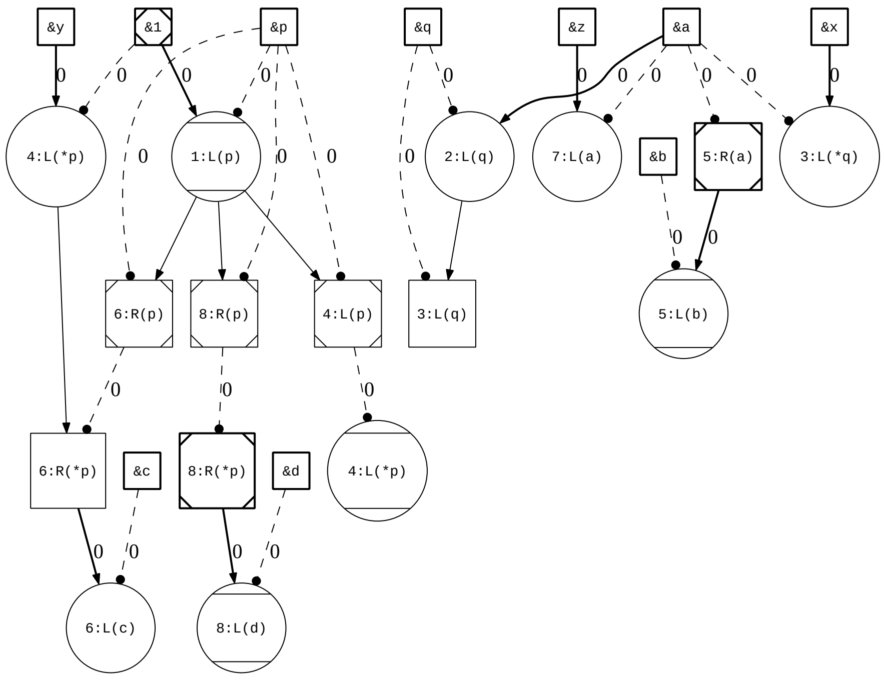
- Let "STORET" be an instruction that modifies an indeterminable address-taken memory location.

The abstraction domain Graph_ϕ is extended to support the two new instruction types.

Let $\mathit{Node}_{\phi_T} = \mathit{ILabel} \cup \{\top\}$ be a set of topped root nodes. If l is a LOADT instruction, then l is represented with the node $l \in \mathit{Node}_{\phi_T}$, and let $\mathit{Pos}_\phi(l) = l$ and $\mathit{ItoN}(l) = l$. The notation $\&k$ is used to refer to a LOADT instruction l, where k is the statement label that contains l.

```
1  p = unknown;
2  q = &a;
3  *q = &x;
4  *p = &y;
5  b = a;
6  c = *p;
7  a = &z;
8  d = *p;
```

(d) Code



(e) Graph

Figure 3.9: Example of store-all nodes

Let "$\top$" be a special topped root node in $Node_{\phi_T}$. If l is a STORET instruction, then l is represented with a dereference node $(l, \top, 0) \in Node_{\phi_D}$. Given $n = (l, \top, 0) \in Node_{\phi_D}$, let $Pos_\phi(n) = l$ and $ItoN(l) = n$. Let $Pos_\phi(\top) = l_{entry}$. The notation $\&k$ is used to refer to a STORET instruction l, where k is the statement label that contains l.

The pseudo-code in Section 3.9 has more details on handling LOADT and STORET instructions.

The graph in Figure 3.9(e) is an abstraction of Figure 3.9(d). Chorded nodes are topped nodes. `4:L(*p)` is a store-all node because `&1` is topped[5]. Since `&a` is address-taken, the store-all node `4:L(*p)` is a possible reaching definition to `5:R(a)`, and thus `5:R(a)` is use-topped and becomes a topped root node. A topped pointer is dereferenced by `6:R(*p)`, but it has a reaching must definition `4:L(*p)` and there are no store-to-address-taken nodes between the must definition and the load, and thus `6:R(*p)` is not use-topped. The store-to-address-taken node `7:L(a)` causes the load node `8:R(*p)` to become use-topped.

3.8 Field-sensitivity

This section introduces field-sensitivity. A flow-sensitive analysis must have some form of field-sensitivity: if the analysis does not differentiate between stores to different offsets in an object, it cannot determine if a value stored in a memory location is overwritten. This section describes field-sensitivity in the abstraction domain *Graph* because field-sensitivity involves reasoning about the values produced by executions, not nMRUs of memory locations. ϕ-nodes have no effect on values.

3.8.1 Relative Offset

We have used slice paths to determine if two nodes may produce an address within the same object. Slice paths can also determine the offset within an object. A STORE instruction l applies its modifier to a value produced by $StVal(l)$ and stores the modified value to memory. Applying a modifier to a value changes the offset within an object that is addressed by the value.

Definition 3.34. Suppose an execution trace tr produces a value (o, f), where o is an object

[5]The promoted store node $($`4:L(*p)`$,$`&1`$, 0)$ replaced the store node $($`4:L(*p)`$,$`4:L(p)`$, 0)$.

$$\mathit{SliceModf} : \mathit{ILabel}^* \to \mathbb{Z}$$

$$\mathit{SliceModf}(tr \cdot [l]) = \begin{cases} 0 & \text{if } l \text{ labels a root instruction} \\ \mathit{SliceModf}(\mathit{LPrefix}(tr, x)) & \text{if } l \text{ labels LOAD } r\ f \\ & \quad \text{where } tr \vdash (\mu, \sigma, \rho, tr) \\ & \quad \text{and } x = \mathit{Last}(\mu(\rho(r) + f)) \\ \mathit{SliceModf}(\mathit{LPrefix}(tr, \mathit{StVal}(l))) & \\ \quad + \mathit{Modf}(l) & \text{if } l \text{ labels a STORE instruction} \end{cases}$$

Figure 3.10: Slice modifier function

and f is an offset. The first instruction in the trace's slice $\mathit{Slice}(tr)$ is the root instruction that produced $(o, 0)$. (Root instructions always produce an address of a memory location with an offset of zero.) The *slice modifier* of tr is the offset f. f can be determined from the slice. Given a trace tr let $\mathit{SliceModf}(tr)$ be tr's slice modifier. Figure 3.10 defines $\mathit{SliceModf}(tr)$.

A set of possible offsets within objects produced by an instruction can be determined in the abstract domain because slices of execution traces abstracted by a graph are represented as paths in the graph. Using this information, nMRUs of different memory locations within an object can be differentiated.

Definition 3.35. Given nodes n and n', the *relative offset set* of a value-aliased node n' relative to n is formed by all integers m such that if n produces an address (o, f) in one execution trace, and n' produces an address (o, f') in another execution trace, then $f - f' = m$.

A function that computes relative offset sets is defined in Figure 3.11. *NModf* maps a node to its modifier. Only store nodes have a non-zero modifier; load nodes and ϕ nodes do not modify values. Given a slice path P, $\mathit{PathModf}(P)$ is its *path modifier*. Given nodes n and n', $\mathit{RelOffs}(G, n, n')$ is n''s relative offset set relative to n.

Given two value-aliased nodes n_s and n'_s, if a use-node n is d-sourced from n_s with a d-offset f, and a def-node n' is d-sourced from n'_s with a d-offset f', then n' is a possible reaching definition to n only if $f' - f$ is in the relative offset set of n'_s relative to n_s.

The graph in Figure 3.12(b) is an abstraction of Figure 3.12(a). The path modifier of [&a, `1:L(p)`, `2:R(p)`, `2:L(q)`, `6:R(q)`], is 1. The path modifier of [&a] is 0. The relative

$$\begin{aligned}
\mathit{Paths} &: \mathit{Graph} \times \mathit{Node} \rightarrow \mathit{Node}^* \\
\mathit{NModf} &: \mathit{Graph} \times \mathit{Node} \rightarrow \mathbb{Z} \\
\mathit{PathModf} &: \mathit{Graph} \times \mathit{Node}^* \rightarrow \mathbb{Z} \\
\mathit{RelOffs} &: \mathit{Graph} \times \mathit{Node} \rightarrow \mathcal{P}(\mathbb{Z})
\end{aligned}$$

$$\mathit{Paths}(G, n, n') = \{[n, \ldots, n'] : [n, \ldots, n'] \text{ is a path in } G\}$$

$$\mathit{NModf}(n) = \begin{cases} \mathit{Modf}(l) & \text{if } n \text{ represents a STORE node } l \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{aligned}
\mathit{PathModf}([n]) &= 0 \\
\mathit{PathModf}(P \cdot [n]) &= \mathit{PathModf}(G, P \cdot [n']) + \mathit{NModf}(n)
\end{aligned}$$

$$\begin{aligned}
\mathit{RelOffs}(G, n, n') = \{m : (&\exists \text{a root node } n_r \in G_N, \\
&P \in \mathit{Paths}(G, n_r, n), P' \in \mathit{Paths}(G, n_r, n') \\
&\mid m = \mathit{PathModf}(P) - \mathit{PathModf}(P'))
\end{aligned}$$

Figure 3.11: Relative offset set

offset set of &a relative to `6:R(q)` is $\{-1\}$.

Suppose a hypothetical def-node n has a d-source &a and d-offset f'. Let f be the d-offset of `6:R(*(q-1))`. If n is a possible reaching definition of `6:R(*(q-1))`, then f' must satisfy $f' - f \in \mathit{RelOffs}(G, \texttt{6:R(q)}, \&a) = \{-1\}$.

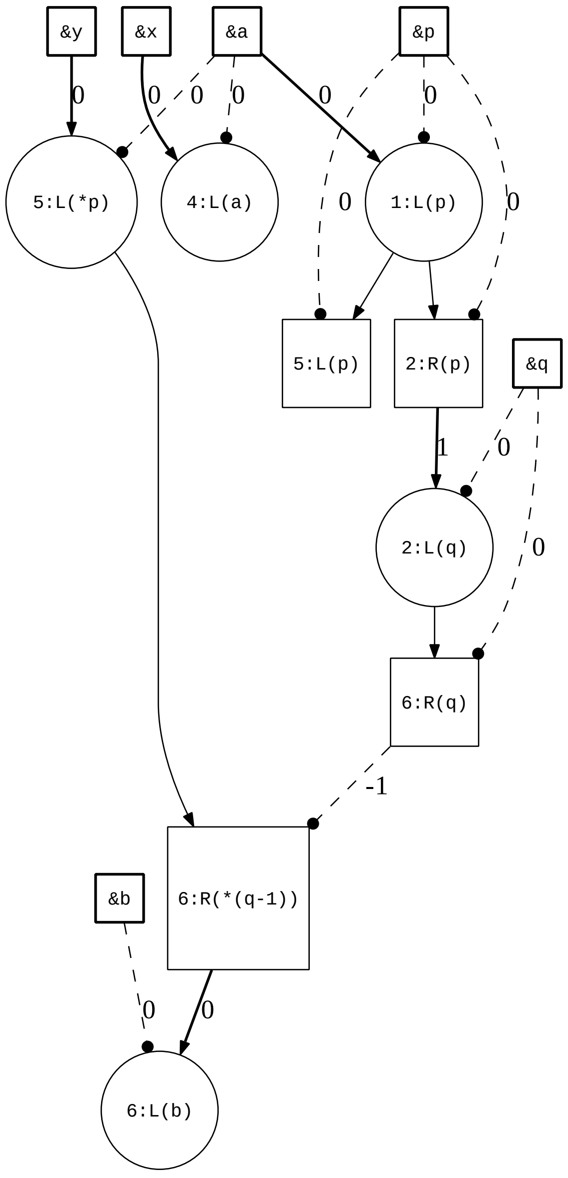
3.8.2 Location Sets

If an input program has loops, then there may be an infinite number of slices and the size of a relative offset set may be infinite. *Location sets* are finite abstractions of a set of integers that can represent relative offset sets. The design of location sets is derived from a paper by Wilson *et al.* [28].

Definition 3.36. A location set consists of a pair of an integer *offset*, and a non-negative integer *stride*. Given an offset f and a stride s, the pair is written $f \pm s$. The location set $f \pm s$ represents the set of integers $\{f + sx : x \in \mathbb{Z}\}$. The sets $\{(f + sx) \pm s : x \in \mathbb{Z}\}$ form an equivalence class of location sets, and the canonical form of a location set is $(f \pmod{s}) \pm s$. A *singular* location set is one that represents a single offset within an object: its stride is

```
1  p = &a;
2  q = p + 1;
3
4  a = &x;
5  *p = &y;
6  b = *(q-1);
```

(a) Code



(b) Graph

Figure 3.12: Example of slice modifiers and relative offsets

$$
\begin{aligned}
(f_1 \pm s_1) + (f_2 \pm s_2) &\equiv ((f_1 + f_2) \pm \gcd(s_1, s_2)) \\
(f_1 \pm s_1) - (f_2 \pm s_2) &\equiv ((f_1 - f_2) \pm \gcd(s_1, s_2)) \\
(f_1 \pm s_1) \sqcup (f_2 \pm s_2) &\equiv (f_1 \pm \gcd(\gcd(s_1, s_2), |f_1 - f_2|)) \\
(f \pm s) + \bot \equiv \bot + (f \pm s) &\equiv (f \pm s) \\
(f \pm s) - \bot &\equiv (f \pm s) \\
\bot - (f \pm s) &\equiv (-f \pm s) \\
(f \pm s) \sqcup \bot \equiv \bot \sqcup (f \pm s) &\equiv (f \pm s)
\end{aligned}
$$

Figure 3.13: Operations on location sets

zero.

The rationale for the design of location sets is to differentiate different fields of aggregates inside an array. Consider the following code snippet:

```
struct  {
    void* p;
    void* q;
} A[10];
```

If the size of pointers is one byte, the memory location referenced by expressions `A[i].p` and `A[i].q`, where `i` is a variable, can be represented by offsets 0 ± 2 and 1 ± 2 within the object `A`, respectively.

To support pointer arithmetic involving non-constants, we redefine the d-offset and modifier of instructions to be location sets. For example, for a C statement "`*(&a+1+i*2) = &b+3+j*4`", where `i` and `j` are variables and all types have a size of one byte, the d-offset of the store node associated with the statement is 1 ± 2 and the modifier of the store node is 3 ± 4.

Definition 3.37. Let $LSet = (\mathbb{Z} \times \mathbb{Z}) \cup \{\bot\}$ be the set of location sets. Location sets are ordered by the subset relation on their representative sets of integers and form a lattice. $\bot$ represents the empty set and the top element of the lattice, $\top$, is the location set 0 ± 1. Addition, subtraction, and join operations on location sets are defined in Figure 3.13. A location set *overlaps* another location set if they represent sets of integers that overlap.

3.8.3 Implementation Details

Suppose that, given an abstraction G, a non-join-point t, a node n and a location set f, we are computing an approximation of $\mathit{ReachDef}(G, t, n, f)$.

A *relative location set* abstracts a relative offset set if the offset set is a subset of the set of integers represented by the location set. The relative location set relative to n of all nodes value-aliased to n is computed. Computing relative location sets is a simple data-flow problem. Given a node n', let $F(n')$ be the meet-over-all-paths solution of path modifiers from root nodes value-aliased with n, to n'. $F(n')$ can be computed with an iterative algorithm. Then, given a node n', $F(n) - F(n')$ is the relative location set of n' relative to n.

The assignment list for a def group with a d-source of n' and a d-offset of f' is stored in a list of assignment lists associated with n'. Let E be the relative location set of n' relative to n. If n' is value-aliased with n, a linear scan of its list of assignment lists is performed to determine the def-groups that have a d-offset that overlaps E. If the d-offset of a def-group N does not overlap E, then no def-node in N is in $\mathit{ReachDef}(G, t, n, f)$.

Small changes are required to handle non-singular d-offsets and modifiers:

- The properties of immediate and residual def-nodes of a def-group hold only if the d-offset of the def-group is singular. If the d-offset of a def-group N is non-singular then all def-nodes that dominates t in N are possible reaching definitions: the immediate node of N is not guaranteed to overwrite other definitions because the def-nodes in N may be storing to different offsets within an object.

- When computing an approximate equal set of a node n, the path modifier from an element of the set to n must be singular. For example, consider the following code segment:

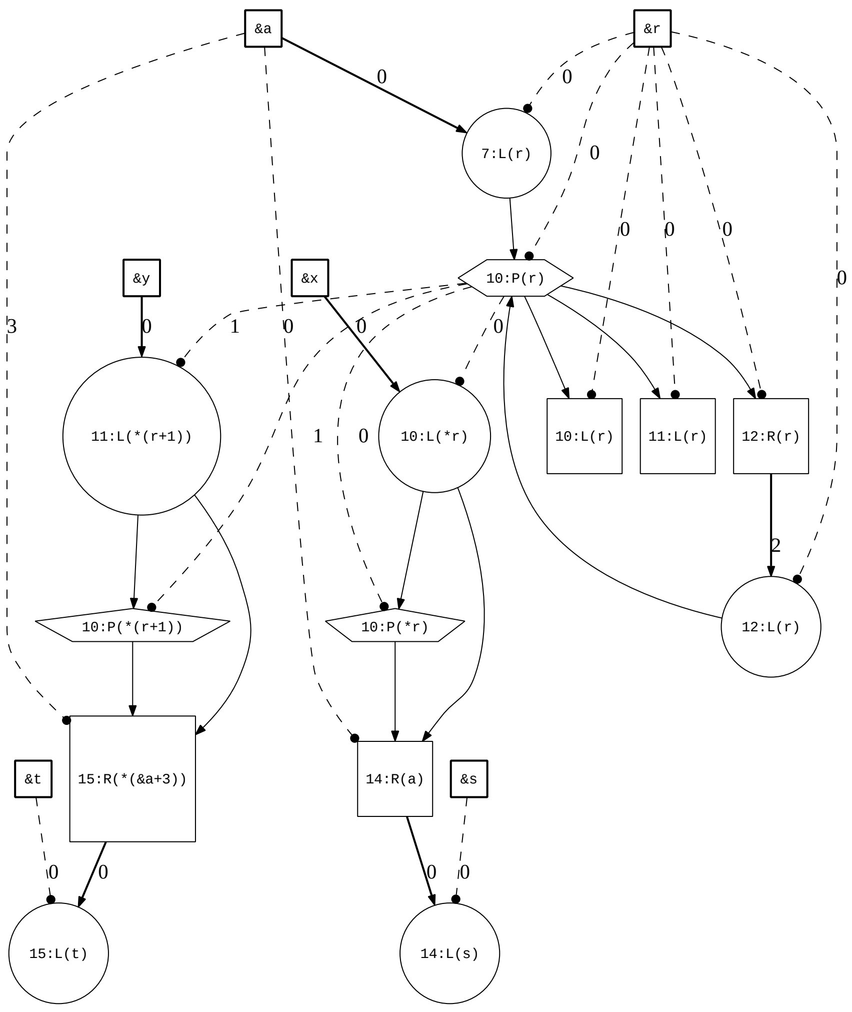
```
1  p = q + i;
2  ... = p;
```

  `1:L(p)` is in the equal set of `2:R(p)`, but `1:R(q)` is not in the equal set of `2:R(p)` because the modifier of `1:R(q)` is non-constant

The graph in Figure 3.14(b) is an abstraction of Figure 3.14(a). The example assumes that the size of all types is one byte. (The C statement `r = r + 1` increments `r` by the

```
1   struct
2   {
3       int* p;
4       int* q;
5   } a[5], *r;
6
7   r = a;
8
9   do {
10      r->p = &x;
11      r->q = &y;
12      r = r + 1;
13  } while( ... );
14  s = a[0].p;
15  t = a[1].q;
```

(a) Code



(b) Graph

Figure 3.14: Example of non-singular location sets

size of the type pointed-to by `r`, which is two bytes.) The set of slice modifiers from `&a` to `l0:P(r)` is infinite in size: $\{0, 2, 4, \ldots\}$. The location set 0 ± 2 abstracts this set. The analysis is able to determine that the store nodes `l0:L(*r)` and `l1:L(*(r+1))`, d-sourced from `l0:P(r)` with d-offsets 0 and 1 respectively, do not store to overlapping regions in an object.

3.9 Flow-sensitive and Field-sensitive Algorithm

This section presents the pseudo-code to our flow-sensitive and field-sensitive algorithm.

The input to the algorithm is a collection of sets and mappings that describe a program:

- *ILabel* is the set of labels of instructions.
- *Instr* is the set of labelled instructions.
- $\mathit{CFlow} \in \mathcal{P}(\mathit{ILabel} \times \mathit{ILabel})$ is the set of control flow edges between instructions.
- *Var* is the set of variables.
- $\mathit{StVal} : \mathit{ILabel} \rightarrow \mathit{ILabel}$ is the mapping from a STORE instruction to its store-value instruction.
- $\mathit{DSrc} : \mathit{ILabel} \rightarrow \mathit{ILabel}$ is the mapping from a STORE or LOAD instruction to its d-source instruction.
- $\mathit{DOff} : \mathit{ILabel} \rightarrow \mathit{LSet}$ is a mapping from a STORE or LOAD instruction to its d-offset.
- $\mathit{Modf} : \mathit{ILabel} \rightarrow \mathit{LSet}$ is a mapping from a STORE instruction to its modifier.
- l_{entry} is the first instruction of the program.

The output of the algorithm is an abstraction of the program in Graph_ϕ.

To differentiate between elements of sets of nodes that are structurally equivalent, when a node is expressed as a tuple, the notation $(v)_A$, $(l, n_s, f)_D$, $(l)_M$, and $(l)_T$ is used to denote an address node, a dereference node, a malloc node, and a topped root node, respectively.

Expressions of the form $\mu[x \mapsto y]$ are transformations of mappings between sets and the notation is described in Section 1.3.

Let $G \in \mathit{Graph}_\phi$, $n \in \mathit{Node}_\phi$, $f \in \mathit{LSet}$, $l \in \mathit{ILabel}$, and $t \in \mathit{ProgPt}$. The algorithm uses the following utility functions.

- $\mathit{CfPriorPt}(t)$ is the set of points immediately prior to t in the control flow graph.
- $\mathit{DfPred}(G, n)$ is the set of nodes that immediately precedes n in the (data-flow) abstraction graph G.
- $\mathit{DfSucc}(G, n)$ is the set of nodes that immediately succeeds n in the abstraction graph G.
- $\mathit{DSrcd}(G, n, f)$ is the set of def-nodes that have d-source n and d-offset f.
- $\mathit{DSrcdF}(G, n)$ is the set of d-offsets of def-nodes that are d-sourced from n.
- $\mathit{NodesAt}(G, l)$ is the set of nodes positioned at l.
- $\mathit{PtAbove}(l)$ is the point above l.
- $\mathit{PtBelow}(l)$ is the point below l.
- $\mathit{InstrAbove}(t)$ is the instruction above t, defined if t is a point below an instruction.
- $\mathit{InstrBelow}(t)$ is the instruction below t, defined if t is a point above an instruction.

The main loop of the analysis is in Figure 3.15. The global variable $G = (G_N, G_E) \in \mathit{Graph}_\phi$ is the abstraction graph that is constructed. The variables $\mathit{maxversion} \in \mathbb{Z}$ and $\mathit{version} : \mathit{ILabel} \to \mathbb{Z}$ ensure that if there is a change in G after processing an instruction l, then all instructions cf-reachable from l is reprocessed.

Procedures in Figures 3.17, 3.18, 3.19, 3.20, 3.22, and 3.23 are procedures that *process* an instruction. Given an instruction l, if G abstracts a trace tr such that $(\mathit{Last}(tr), l) \in \mathit{CFlow}$, then G also abstracts $tr \cdot [l]$ after processing l. Each procedure returns *true* if G changed after processing an instruction. ProcessSTORE and ProcessLOAD also return a second boolean value that specifies if an uninitialized value is being dereferenced in all traces abstracted by G. According to the concrete semantics, a trace that dereferences an uninitialized value

Algorithm 3.9.1: MAIN()

```
global $G = (G_N, G_E) \in Graph_\phi$
global $CfReached \in \mathcal{P}(ILabel)$
local $l, l' \in ILabel$
local $v \in Var$
local $version : ILabel \to \mathbb{Z}$
local $maxversion \in \mathbb{Z}$
local $worklist \in \mathcal{P}(ILabel)$
local $changed, nostate : Boolean$

Compute dominator tree
Compute dominance frontier closures
$version \leftarrow \lambda\, x\, .\, 0$
$maxversion \leftarrow 1$
$version \leftarrow version[l_{entry} \mapsto maxversion]$
$worklist \leftarrow \{l_{entry}\}$
for each $v \in Var$
  do $nodes \leftarrow nodes \cup \{(v)_A\}$
while $worklist$ is not empty
  do
    Pop $l$ from $worklist$
    $changed \leftarrow false$
    if $l$ labels a LOADM instruction
      then $changed \leftarrow$ PROCESSLOADM($l$)
      else if $l$ labels a LOAD instruction
      then $(changed, nostate) \leftarrow$ PROCESSLOAD($l$)
      else if $l$ labels a STORE instruction
      then $(changed, nostate) \leftarrow$ PROCESSSTORE($l$)
      else if $l$ labels a STORET instruction
      then $changed \leftarrow$ PROCESSSTORET($l$)
      else if $l$ labels a LOADT instruction
      then $changed \leftarrow$ PROCESSLOADT($l$)
      else if $l$ labels a SKIP instruction
      then $changed \leftarrow$ PROCESSSKIP($l$)
    if $changed$
      then
        $maxversion \leftarrow maxversion + 1$
        $version \leftarrow version[l \mapsto maxversion]$

    if $\neg nostate$
      then
        $CfReached \leftarrow CfReached \cup \{l\}$
        for each $\{l' : (l, l') \in CFlow\}$
          do
            if $version(l') < version(l)$
              then
                $version \leftarrow version[l' \mapsto version(l)]$
                $worklist = worklist \cup \{l'\}$
```

Figure 3.15: Main loop

```
Algorithm 3.9.2: CleanAndAdd($l \in ILabel$,
$newNodes \in \mathcal{P}(Node_\phi), newEdges \in \mathcal{P}(Edge_\phi)$)

  global $G = (G_N, G_E) \in Graph_\phi$
  local $oldNodes \in \mathcal{P}(Node_\phi)$
  local $oldEdges \in \mathcal{P}(Edge_\phi)$

  $oldNodes \leftarrow NodesAt(G, l)$
  $oldEdges \leftarrow \{(n, n') \in G_E, n \in NodesAt(G, l) \vee n' \in NodesAt(G, l)\}$
  $G_N \leftarrow (G_N \setminus oldNodes) \cup newNodes$
  $G_E \leftarrow (G_E \setminus oldEdges) \cup newEdges$
  return $(oldNodes \neq newNodes \vee oldEdges \neq newEdges)$
```

Figure 3.16: Committing changes to an abstraction

does not reach a program state. Thus, the analysis does not process instructions that are dominated by an instruction that dereferences an uninitialized value.

CfReached records whether G abstracts a trace that reaches a particular instruction. If $l \in CfReached$, then G abstracts a trace that reaches $PtBelow(l)$. This information is used by ProcessSKIP to determine the points at which the set of possible reaching definitions of a ϕ node should be computed.

When an instruction is processed, nodes that must be created at the position of the instruction and incoming edges of the created nodes are added to the abstraction. The procedure in Figure 3.16 applies changes to the abstraction. If the instruction is being reprocessed, then nodes and edges added to the abstraction when the instruction was previously processed is removed from the abstraction. If the set of nodes and edges that are added to the abstraction is equal to the set of nodes and edges that are removed, then no change has been made to G.

The procedure in Figure 3.17 processes a LOAD instruction. If l labels a LOAD instruction, then there is only one node at l during the entire analysis. CleanAndAdd is used for convenience.

To compute the set of possible reaching definitions of the load node, the procedure ComputeReachDef is invoked, which is listed in Figure 3.25. ComputeReachDef returns a pair: a conservative set of possible reaching definitions and a boolean specifying whether the load node has use-topped.

Algorithm 3.9.3: PROCESSLOAD($l \in ILabel$)

```
  global $G = (G_N, G_E) \in Graph_\phi$
  local $n, n_s \in Node_\phi$
  local $newEdges \in \mathcal{P}(Edge_\phi)$
  local $reachDefs \in \mathcal{P}(Node_\phi)$
  local $isUseTopped : Boolean$

  $n_s \leftarrow ItoN(G, DSrc(l))$
  $n \leftarrow (l, n_s, DOff(l))_D$

  if $n_s$ is not df-reachable from a root node
    then return $(false, true)$

  $(reachDefs, isUseTopped)$
        $\leftarrow$ COMPUTEREACHDEF($PtAbove(l), n_s, DOff(l)$)

  if $isUseTopped$
    then $n \leftarrow (l)_T$

  $newEdges \leftarrow \{(n', n) : n' \in reachDefs\}$
  return (CLEANANDADD($l, \{n\}, newEdges$), $false$)
```

Figure 3.17: Processing a LOAD instruction

Algorithm 3.9.4: PROCESSLOADM($l \in ILabel$)

```
  global $G = (G_N, G_E) \in Graph_\phi$
  local $n \in Node_\phi$

  $n \leftarrow (l)_M$
  if $n \notin G_N$
    then { $G_N \leftarrow G_N \cup \{n\}$
         { return $(true)$
    else return $(false)$
```

Figure 3.18: Processing a LOADM instruction

Algorithm 3.9.5: PROCESSLOADT($l \in ILabel$)

```
global $G = (G_N, G_E) \in Graph_\phi$
local $n \in Node_\phi$

$n \leftarrow (l)_T$
if $n \notin G_N$
  then $\begin{cases} G_N \leftarrow G_N \cup \{n\} \\ \textbf{return}\ (true) \end{cases}$
  else return $(false)$
```

Figure 3.19: Processing a LOADT instruction

Algorithm 3.9.6: PROCESSSTORET($l \in ILabel$)

```
global $G = (G_N, G_E) \in Graph_\phi$
local $n \in Node_\phi$

$n \leftarrow (l, \top, 0)_D$
if $n \notin G_N$
  then $\begin{cases} G_N \leftarrow G_N \cup \{n\} \\ \textbf{return}\ (true) \end{cases}$
  else return $(false)$
```

Figure 3.20: Processing a STORET instruction

Algorithm 3.9.7: ComputeEqualSet($n \in Node_\phi$)

```
global $G = (G_N, G_E) \in Graph_\phi$
global $oldEqSets : Node_\phi \to \mathcal{P}(Node_\phi)$
local $n', n'' \in Node_\phi$
local $eqSet \in \mathcal{P}(G_N)$

$n' \leftarrow n$
$eqSet \leftarrow \{n\}$
while $n'$ has a single df-predecessor $n''$ in $G$ such that
    $n''$ dominates $n'$,
    $n''$'s d-source node dominates $n''$,
    and $NModf(n'')$ is singular
  do $\begin{cases} n' \leftarrow n'' \\ eqSet \leftarrow eqSet \cup \{n''\} \end{cases}$

if $oldEqSets(n) \neq \emptyset$
  then $eqSet \leftarrow eqSet \cap oldEqSets(n)$
$oldEqSets \leftarrow oldEqSets[n \mapsto eqSet]$
return $(n')$
```

Figure 3.21: Computing a conservative equal set

The procedure in Figures 3.18, 3.19, and 3.20 processes a LOADM instruction, a LOADT instruction, and a STORET instruction, respectively.

An approximation of an equal set is computed by the procedure in Figure 3.21. Given a node n, a conservative equal set is a subset of $EqualSet(G, n)$. To ensure that the algorithm terminates, the equal set approximation is forced to monotonically shrink in size. For all nodes n, n is always in the approximate equal set of n. Therefore, the empty set is used as an uninitialized value for *oldEqSets*.

The procedure in Figure 3.22 processes a STORE instruction. If l labels a STORE instruction, then there is only one node at l during the entire analysis. However, the d-source of a store node positioned at l may change during the analysis, due to changes to the equal set approximation of the node that represents $DSrc(l)$.

The procedure in Figure 3.23 processes a SKIP instruction. Given an instruction l, the procedure iterates through all def-nodes n that may induce a ϕ node at l. Computing the set of possible reaching definitions of a ϕ node is a process that is similar to processing a LOAD node, except that sets of possible reaching definitions are computed at points that

```
Algorithm 3.9.8: ProcessSTORE($l \in ILabel$)

  global $G = (G_N, G_E) \in Graph_\phi$
  local $n, n', n_s \in Node_\phi$
  local $(l', n'_s, f')_D \in Node_{\phi D}$
  local $f \in LSet$
  local $eqSet \in \mathcal{P}(Node_\phi)$
  local $newEdges \in \mathcal{P}(Edge_\phi)$

  if $ItoN(G, DSrc(l))$ is not df-reachable from a root node
    then return $(false, true)$

  $eqSet \leftarrow$ ComputeEqualSet($ItoN(G, DSrc(l))$)

  $n_s \leftarrow$ the node that dominates all other nodes in $eqSet$
  $f \leftarrow PathModf$(df-path from $ItoN(G, DSrc(l))$ to $n_s$)
  $n \leftarrow (l, n_s, DOff(l) + f)_D$

  $newEdges = \{(ItoN(G, StVal(l)), n)\}$

  return (CleanAndAdd($l, \{n\}, newEdges$), $false$)
```

Figure 3.22: Processing a STORE instruction

are immediately prior to the join-point above l in the control flow graph.

If l is a SKIP instruction, then there may be multiple immediate predecessors of l in the control flow graph. When l is processed, the abstraction is not guaranteed to abstract a trace that ends at each of the instructions that are immediately prior to l. If t' is a point immediately prior to the join-point above l, then a set of possible reaching definitions is computed at t' only if the abstraction abstracts a trace that ends at the instruction above t'.

Given a node n, the procedure in Figure 3.24 computes an approximation of the relative location sets of each node that is value-aliased with n, relative to n. The computation is performed in two steps: first, the backward phase computes the relative location sets of each root node that is value-aliased with n; then, the forward phase computes the relative location sets of all value-aliased nodes. During each phase, the set of integers represented by the approximate relative location set of a node monotonically increases ($\perp$ represents the empty set). Thus, if the stride of an approximate relative location set is non-zero, the stride monotonically decreases. Therefore, each phase terminates for all inputs because the stride

Algorithm 3.9.9: ProcessSKIP($l \in ILabel$)

```
global $G = (G_N, G_E) \in Graph_\phi$
global $CfReached \in \mathcal{P}(ILabel)$
local $n \in Node_\phi$
local $(l', n'_s, f')_D \in Node_{\phi D}$
local $(l'', n''_s, f'')_D \in Node_{\phi D}$
local $newNodes \in \mathcal{P}(Node_\phi)$
local $newEdges \in \mathcal{P}(Edge_\phi)$
local $reachDefs, reachDefsTmp \in \mathcal{P}(Node_\phi)$
local $isUseTopped, isUseToppedTmp : Boolean$


for each $(l', n'_s, f')_D \in ItoN_\phi(G, l)$
  do
    $n \leftarrow (l, n'_s, f')_D$
    for each $t' \in CfPriorPt(PtAbove(l))$
               $: InstrAbove(t') \in CfReached$
      do
        $(reachDefsTmp, isUseToppedTmp)$
             $\leftarrow$ ComputeReachDef$(t', n'_s, f')$

        // If $(l, n'_s, f')_D$ is a weak $\phi$ node,
        // remove def-nodes that are not in the same
        // def-group as $(l, n'_s, f')_D$
        if $n'_s$ does not dominate $l$
          then $reachDefsTmp \leftarrow$
                 $\{(l'', n''_s, f'')_D \in reachDefsTmp : n''_s = n'_s\}$

        $reachDefs \leftarrow reachDefs$
             $\cup reachDefsTmp$
        $isUseTopped \leftarrow isUseTopped$
             $\vee isUseToppedTmp$

    if $isUseTopped$
      then $n \leftarrow (l)_T$
    $newNodes \leftarrow newNodes \cup \{n\}$
    $newEdges \leftarrow newEdges \cup \{(n', n) : n' \in reachDefs\}$
return (CleanAndAdd$(l, newNodes, newEdges)$)
```

Figure 3.23: Processing a SKIP instruction

Algorithm 3.9.10: COMPUTERELATIVELSET($t \in \mathit{ProgPt}, n \in \mathit{Node}_\phi$)

```
global $G = (G_N, G_E) \in \mathit{Graph}_\phi$
local $\mathit{rootRelLSet} : \mathit{Node}_\phi \to \mathit{LSet}$
local $\mathit{relLSet} : \mathit{Node}_\phi \to \mathit{LSet}$
local $\mathit{lset} \in \mathit{LSet}$
local $\mathit{worklist} \in \mathcal{P}(\mathit{Node}_\phi)$

$\mathit{rootRelLSet} \leftarrow \lambda\, x\, .\, \bot$
$\mathit{rootRelLSet} \leftarrow \mathit{rootRelLSet}[n \mapsto 0 \pm 0]$
$\mathit{worklist} \leftarrow \mathit{DfPred}(n)$
while $\mathit{worklist}$ is not empty
  do
    Pop $n'$ from $\mathit{worklist}$
    $\mathit{lset} \leftarrow \bigsqcup\{\mathit{rootRelLSet}(n'') + \mathit{NModf}(n') : n'' \in \mathit{DfSucc}(n'')\}$
    if $\mathit{lset} \neq \mathit{rootRelLSet}(n')$
      then
        $\mathit{rootRelLSet} \leftarrow \mathit{rootRelLSet}[n' \mapsto \mathit{lset}]$
        $\mathit{worklist} \leftarrow \mathit{worklist} \cup \mathit{DfPred}(n')$

$\mathit{relLSet} \leftarrow \lambda\, x\, .\, \bot$
for each root node $n'$ in $\mathit{ValAlias}(G, n)$
  do
    $\mathit{relLSet} \leftarrow \mathit{relLSet}[n' \mapsto \mathit{rootRelLSet}(n')]$
    $\mathit{worklist} \leftarrow \mathit{worklist} \cup \mathit{DfSucc}(n')$

while $\mathit{worklist}$ is not empty
  do
    Pop $n'$ from $\mathit{worklist}$
    $\mathit{lset} \leftarrow \bigsqcup\{\mathit{relLSet}(n'') - \mathit{NModf}(n'') : n'' \in \mathit{DfPred}(n'')\}$
    if $\mathit{lset} \neq \mathit{relLSet}(n')$
      then
        $\mathit{relLSet} \leftarrow \mathit{relLSet}[n' \mapsto \mathit{lset}]$
        $\mathit{worklist} \leftarrow \mathit{worklist} \cup \mathit{DfSucc}(n')$
return ($\mathit{relLSet}$)
```

Figure 3.24: Computing a conservative relative location set mapping

is finite.

Given a non-join point t, a node n, and an offset f, the procedure in Figure 3.24 computes a superset of $\mathit{ReachDef}(G, t, n, f)$, which is a conservative set of possible reaching definitions for memory locations specified by n and f (the *loaded* memory locations).

The procedure obtains the relative location sets of all value-aliased nodes. Then, for each node n' that is value-aliased with n, def-nodes that are d-sourced from n' are grouped by their d-offsets to form a def-group N. In the actual implementation, def-groups are maintained in assignment lists [29], and the list is updated whenever a def-node is added or removed from G. Then, the immediate node of a def-group at a point can be queried efficiently.

The immediate and residual def-nodes of N are considered to be possible reaching definitions if N's d-offset overlaps the relative location set of N's d-source node.

An approximation of the equal set of n is obtained and each def-node in the set of possible reaching definitions is checked if it meets the criteria of a must definition.

After must definitions are determined, IsLoadTopped is called to determine whether obtaining precise pointer information is not possible due to the effects of unanalyzable constructs. If unanalyzable constructs have no effect on the reaching definition calculation, then the set of possible reaching definitions are pruned to remove def-nodes that strictly dominate a must definition. The resulting set of possible reaching definitions is returned by the procedure.

The procedure in Figure 3.26 determines if a must definition occludes the effect of unanalyzable constructs. The sets of def-nodes $N_\top$ and N_{ad} are not def-groups because they are not sets of def-nodes with common d-sources and d-offsets. However, they are an union of def-groups, and due to the dominating def-node property, if there is an n-trace from a def-node n' in one of $N_\top$ or N_{ad}, and n_i is the immediate def-node of the set of def-groups at t, then the n-trace must pass through n_i or n' is equal to n_i. The specific memory location defined by n' or n_i do not matter: what matters is whether n_i is a store-to-address-taken node (n_i's d-source is address-taken) or n_i is a store-all node (n_i's d-source is topped).

Algorithm 3.9.11: COMPUTEREACHDEF($t \in \mathit{ProgPt}, n \in \mathit{Node}_\phi, f \in \mathit{LSet}$)

```
global $G = (G_N, G_E) \in \mathit{Graph}_\phi$
local $N, \mathit{reachDefs}, \mathit{eqSet}, \mathit{mustDefs} \in \mathcal{P}(\mathit{Node}_\phi)$
local $\mathit{relLSet} : \mathit{Node}_\phi \to \mathit{LSet}$
local $n', n'', n_i \in \mathit{Node}_\phi$
local $\mathit{immMustDef} \in (\mathit{Node}_\phi \cup \{\mathit{null}\})$

$\mathit{relLSet} \leftarrow$ COMPUTERELATIVELSET($n$)

for each $n' \in \mathit{ValAlias}(G, n)$
  do
    for each $f' \in \{f' \in \mathit{DSrcdF}(G, n')$
                  : $(f' - f)$ overlaps $\mathit{relLSet}(n')$ }
      do
        $N \leftarrow \mathit{DSrcd}(G, n', f')$
        if $f'$ is singular
          then
            if $N$ has an immediate node $n_i$ at $t$
              then $\mathit{reachDefs} \leftarrow \mathit{reachDefs} \cup \{n_i\}$
            if $N$ has an immediate node $n_i$ at $\mathit{PtAbove}(n')$
              then $\mathit{reachDefs} \leftarrow \mathit{reachDefs} \cup \{n_i\}$
          else
            $\mathit{reachDefs} \leftarrow \mathit{reachDefs} \cup \{n'' \in N$
                                   : $n''$ dominates $t\}$

$\mathit{eqSet} \leftarrow$ COMPUTEEQUALSET($n$)

$\mathit{immMustDef} \leftarrow \mathit{null}$
$\mathit{mustDefs} \leftarrow \emptyset$
if f is singular
  then
    for each $n' \in \mathit{eqSet}$
      do
        $N \leftarrow \mathit{DSrcd}(G, n', f)$
        if $N$ has an immediate node $n_i$ at $t$
          then $\mathit{mustDefs} \leftarrow \mathit{mustDefs} \cup \{n_i\}$
if $\mathit{mustDefs} \neq \emptyset$
  then
    $\mathit{immMustDef} \leftarrow$ the immediate node of $\mathit{mustDefs}$
    $\mathit{reachDefs} \leftarrow \{n' \in \mathit{reachDefs}$ :
                           $n'$ does not strictly dominate $\mathit{immMustDef}\}$
if ISLOADTOPPED($t, \mathit{immMustDef}$)
  then return $(\{\}, \mathit{true})$
  else return $(\mathit{reachDefs}, \mathit{false})$
```

Figure 3.25: Computing a conservative set of possible reaching definitions

Algorithm 3.9.12: IsLoadTopped($t \in ProgPt$, $immMustDef \in (Node_\phi \cup \{null\})$)

```
if $n$ is address-taken
  then
    $N_\top \leftarrow \bigcup\{DSrcdF(G, n') : n'$ is topped$\}$
    if $N_\top$ has an immediate node $n_i$ at $t$
      then
        if $immMustDef = null \lor n_i$ does not strictly dominate $immMustDef$
          then return ($true$)

if $n$ is topped
  then
    $N_{ad} \leftarrow \bigcup\{DSrcdF(G, n') : n'$ is address-taken$\}$
    if $N_{ad}$ has an immediate node $n_i$ at $t$
      then
        if $immMustDef = null \lor n_i$ does not strictly dominate $immMustDef$
          then return ($true$)
return ($false$)
```

Figure 3.26: Determining whether a use-node is use-topped

3.10 Termination

Proving that the algorithm terminates for all inputs is complicated by the fact that reprocessing an instruction may change existing store and ϕ nodes in an abstraction. In particular, the d-source node of a store node may change during the analysis. If we can prove that the set of store nodes reaches a fixed point, then the set of ϕ nodes will also reach a fixed point because they are induced by store nodes. After the set of nodes reaches a fixed point, we prove that the number of edges in an abstraction does not decrease when an instruction is reprocessed. That proves that the algorithm terminates for all inputs, because the number of possible edges is finite.

Whenever a STORE instruction is reprocessed, the equal set approximation of the node associated with the d-source of the STORE instruction is a subset of the equal set approximation that was computed when the STORE instruction was previously processed (Figure 3.21). Thus, there is a finite number of times that a store node for a given STORE instruction is replaced with a store node with a different d-source node. Thus, the set of store nodes of an abstraction reaches a fixed point after processing a finite number of instructions.

The set of ϕ nodes is dependent only on the set of store nodes. Thus, once all SKIP instructions have been reprocessed after the set of store nodes has reached a fixed point, the set of ϕ nodes has reached a fixed point. The set of load nodes do not change after all reachable LOAD instructions have been processed. Thus, the set of nodes reaches a fixed point after processing a finite number of instructions.

Suppose that the algorithm is computing an abstraction of a program, and the set of nodes has reached a fixed point. Furthermore, suppose that each instruction has been reprocessed at least once more after the set of nodes has reached a fixed point. Then, for any point t, node n, and offset f, the size of the set returned by ComputeReachDef(t, n, f) does not decrease after reprocessing an instruction:

- The compositions of def-groups do not change; thus the immediate node at t and the residual node of a def-group do not change.

- The approximate equal set of n computed for the purpose of finding must definitions must be a subset of the previous computation of the approximate equal set; thus def-nodes that were not excluded by a must definition when the instruction was previously processed, are not excluded when the instruction is reprocessed either.

Thus, the size of the set returned by ComputeReachDef(t, n, f), and thus the number of edges of an abstraction, does not decrease after reprocessing an instruction. Therefore, the abstraction reaches a fixed point after processing a finite number of instructions.

This chapter described our flow- and field-sensitive algorithm. In the next chapter, the output of our algorithm is evaluated against slightly modified versions of our algorithm that simulate the precision of a points-to-set-based analysis.

Chapter 4

Experimental Evaluation

This chapter explores the performance and precision characteristics of our flow-sensitive and field-sensitive algorithm.

4.1 Methodology

The current analysis is strictly intraprocedural: it is unable to pass pointer information between procedure boundaries. To conservatively analyze all programs, the most pessimistic assumptions must be made:

- Procedure-call statements are treated as unknown statements (store-alls).
- At the entry point of a procedure, all address-taken memory locations and formal parameters are assumed to potentially contain the address of any address-taken memory location (topped).

If the analysis is performed with the above assumptions, then nearly all nodes are topped. Our algorithm is hypothetically more precise than a points-to-set-based analysis, because the algorithm may perform what amounts to a strong update of a non-singular abstract object. If nearly all nodes are topped, the difference in precision between configurations is difficult to measure. To obtain a rough measure of the potential precision benefits between our algorithm and a points-to-set-based analysis, we also ran our algorithm with several optimistic and unsound assumptions:

- Procedure-calls do not alter any memory location.
- Each formal parameters points to a unique object, unaliased with any other object.

- Global variables are uninitialized at the entry point of a procedure.

To simulate the precision of a points-to-set-based analysis, we ran our algorithm in three configurations:

- *Full* - The algorithm as described by this thesis with full flow-sensitivity.
- *No-must-defs* (NMD) - The algorithm does not use must definitions to exclude possible reaching definitions.
- *No-elevate* (NEL) - In addition to NMD, the algorithm does not replace store nodes with promoted store nodes.

Under the NEL configuration, the only def-groups that contain more than one store node are d-sourced from address nodes. The NEL configuration simulates the precision of a points-to-set-based analysis that does not perform indirect strong updates. Points-to-set-based analyses may perform an indirect strong update through a pointer that points to exactly one singular abstract object, and thus the precision of NEL is slightly worse than that of a points-to-set-based analysis.

An artificial metric measures precision. *Indirection removal* is a common optimization that uses pointer information: if a dereference expression is aliased with a single local variable, then the expression can be replaced with the variable: for example, if “p” must point to “a” when evaluated at an expression “*p” in a program, then “*p” can be replaced with “a”.

We propose a slightly more general optimization: a load node is said to be *replaceable* if it has a single reaching definition. The idea is that if a load node represents a sub-expression e and has a single reaching definition n, then a store to a temporary scalar variable can be inserted below n's position, and e can be replaced by a direct reference to the scalar variable. For example, consider the listing below:

```
p = ... ? &a : &b;
*p = x;

...
... = *p;
```

If `2:L(*p)` is the only reaching definition of `5:R(*p)`, then the code can be transformed into the following:

```
1 p = ... ? &a : &b;
2 *p = x;
3 tmp = x;
4 ...
5 ... = tmp;
```

Thus, if a load node is replaceable, then the back-end has the option to remove an access to memory. The transformation may be detrimental to performance because it increases registry pressure, but having the option to perform the transformation is never detrimental.

ϕ nodes affect whether a load node is replaceable. A ϕ node is replaceable if each point immediately prior to the join-point above the position of the ϕ node has at most one reaching definition that dominates the point. For example, consider the listing below:

```
1 if(...) {
2     *p = x;
3 
4 }
5 else {
6     *p = y;
7 
8 }
9 ... = *p;
```

The reaching definition of the load node `9:R(*p)` is a ϕ node. The ϕ node has two possible reaching definitions: a reaching definition each from the two incoming control flow paths. Thus, the ϕ node is replaceable: if all its possible reaching definitions store to the same temporary variable, the temporary variable has the same value as the memory location stored to by the possible reaching definitions:

```
1 if(...) {
2     *p = x;
3     tmp = x;
4 }
5 else {
6     *p = y;
7     tmp = y;
8 }
9 ... = tmp;
```

The full requirements for a load node to be replaceable is listed below:

Name	SLOC	Stores	Loads	Prelim	Memory
povray	78705	50839	60650	0.58s	14MB
perlbench	126245	33029	41835	0.43s	12MB
gobmk	157654	38994	44016	0.87s	13MB
xalancbmk	269176	270382	336706	6.26s	110MB
gcc	382843	152441	163638	2.47s	67MB

Table 4.1: Benchmark metrics

- The load node has a single reaching definition.
- For all df-paths that passes only through ϕ nodes from a store node to the load node, all ϕ nodes within the df-paths must be replaceable.

Our implementation was compiled using GCC 4.5 with the `-O3` optimization flag. The front-end of the analysis is an IBM XL compiler, which converts C code to a proprietary intermediate representation. Our analysis parses the intermediate representation into its own representation in memory. We ran the analysis on a machine with an Intel Core 2 Duo E6300 processor with 2GB of RAM running 32-bit Ubuntu 11.04. The running times are an average of three runs.

4.2 Results

The analysis was performed on the five largest CPU2006 C/C++ benchmarks. The benchmarks are described in Table 4.1. **SLOC** is the number of source lines of C code[1]. **Stores** is the number of store nodes, which corresponds to the number of *assignment expressions*. In C, a single statement may contain multiple assignment expressions by chaining, e.g. "`p = q = r;`", or by sequencing, e.g. "`(q = r), (p = q);`". **Loads** is the number of load nodes. Only expressions that appear in the right-hand side of an assignment expression is represented by a load node; other use expressions have no effect on a program state. **Prelim** is the time in seconds taken by the analysis to read the source files and to compute the dominator tree and dominance frontier sets of every procedure. **Memory** is the amount of RAM in MB used by the analysis program to hold the input representation.

Table 4.2 lists precision metrics for the Full, NMD, and NEL configurations under pessimistic assumptions. **Ld** is the number of non-root and non-topped load nodes. **Rp** is the

[1]The number of source lines of C code is measured by David Wheeler's SLOCCount program version 2.26.

	Ld	**Rp**	**NdLd**	**NdRp**	**Time**	**Memory**
povray (Full)	5093	3994	45	44	0.34s	56MB
povray (NMD)	5048	3950	0	0	0.34s	56MB
povray (NEL)	5048	3950	0	0	0.32s	56MB
perlbench (Full)	697	344	7	7	0.35s	48MB
perlbench (NMD)	690	337	0	0	0.34s	48MB
perlbench (NEL)	690	337	0	0	0.33s	48MB
gobmk (Full)	1798	322	3	2	4.71s	49MB
gobmk (NMD)	1795	320	0	0	4.18s	49MB
gobmk (NEL)	1795	320	0	0	2.28s	50MB
xalancbmk (Full)	7202	4715	91	91	0.76s	322MB
xalancbmk (NMD)	7135	4650	38	38	0.59s	322MB
xalancbmk (NEL)	7097	4612	0	0	0.57s	322MB
gcc (Full)	7157	4649	29	29	1.24s	224MB
gcc (NMD)	7127	4620	0	0	1.19s	224MB
gcc (NEL)	7127	4620	0	0	1.14s	224MB

Table 4.2: Pessimistic assumptions

number of replaceable non-root and non-topped load nodes. Let *direct nodes* be nodes that are d-sourced from address nodes i.e. the nodes represent expressions that are direct references to variables. **NdLd** is the number of non-direct, non-root, and non-topped load nodes. **NdRp** is the number of replaceable non-direct, non-root, and non-topped load nodes.

Due to the frequent occurrence of store-alls, there are almost no non-direct load nodes that are not topped. A few non-direct and non-topped load nodes exist in the Full configuration. Since only a must definition can occlude the effects of a store-all, the few non-direct and non-topped load nodes nearly disappear in the NMD and NEL configurations.

Table 4.3 lists precision metrics for the Full, NMD, and NEL configurations under optimistic assumptions. The difference in the number of replaceable non-direct load nodes (column **NdRp**) between the Full configuration and the NEL configuration is the best indicator of the precision difference of our algorithm and a points-to-set-based analysis. The number of replaceable non-direct load nodes decreases by 45%, 58%, 55%, 46%, and 26% between the Full configuration and NEL configuration for the “povray”, “perlbench”, “gobmk”, “xalancbmk”, and “gcc” benchmarks, respectively.

The numbers of replaceable non-direct nodes of the Full and NMD configurations are nearly the same. This may indicate that there are infrequent occurrences of code segments that have interleaved indirect stores to a common memory location through potentially dif-

	Ld	**Rp**	**NdLd**	**NdRp**	**Time**	**Memory**
povray (Full)	29094	20046	6121	1144	0.33s	51MB
povray (NMD)	29094	20023	6121	1121	0.33s	51MB
povray (NEL)	29094	19460	6121	624	0.40s	51MB
perlbench (Full)	22972	11378	7190	547	0.42s	47MB
perlbench (NMD)	22972	11378	7190	547	0.40s	47MB
perlbench (NEL)	22972	11061	7190	230	0.41s	47MB
gobmk (Full)	12335	4154	829	55	3.45s	48MB
gobmk (NMD)	12335	4153	829	54	3.92s	48MB
gobmk (NEL)	12335	4124	829	25	10.94s	53MB
xalancbmk (Full)	145195	109120	29937	4461	0.59s	266MB
xalancbmk (NMD)	145195	109106	29937	4447	0.58s	266MB
xalancbmk (NEL)	145195	107049	29937	2417	0.58s	266MB
gcc (Full)	92928	42992	32693	1367	3.14s	217MB
gcc (NMD)	92928	42880	32693	1255	3.14s	217MB
gcc (NEL)	92928	42637	32693	1012	3.07s	217MB

Table 4.3: Optimistic assumptions

ferent pointers. As an example, consider the following listing:

```
1  p = ... ? &a : &b;
2  q = ... ? &a : &b;
3  *p = ...;
4  *q = ...;
5  ... = *q;
```

The store node `4:L(*q)` is a must definition to the load node `5:R(*q)`, and excludes the store node `3:L(*p)`. In the NMD and NEL configurations, `5:R(*q)` have two possible reaching definitions and is irreplaceable.

In Table 4.3, the running time of the analysis for the benchmark “gobmk” shows a large increase over other configurations that is unalike other benchmarks. The analysis spends the most time analyzing a particular procedure that makes heavy use of a C macro that expands into a deeply nested block of code that has several dozen indirect loads and stores. This results in the procedure having 5257 ϕ nodes induced by 795 store nodes, for a total of 6426 use-nodes. In the Full configuration, each use-node has an average of about 2 possible reaching definitions. In the NEL configuration, each use-node has an average of about 17 possible reaching definition, which is likely the cause of the slowdown.

The next chapter explores related works. One paper in particular attempts to improve the precision of pointer information by identifying expressions that evaluate to the same value, which is related to how our analysis expresses data dependencies directly between expressions [4]. This paper is compared to our approach in detail.

Chapter 5

Related Work

Pointer analyses can be differentiated into three categories. A points-to analysis partitions the infinite number of run-time locations into a finite number of abstract objects, and computes points-to relations between abstract objects. An *alias analysis* computes alias relations between pairs of program expressions [6, 8]. A *shape analysis* attempts to discover properties of dynamically allocated data structures, such as whether a linked list is acyclic or whether nodes of two linked lists are aliased pairwise [20].

Most recent papers on pointer analysis have used the points-to abstraction since it is a compact representation of alias relations. For example, the points-to relation (`p`, `a`), represents an unbounded number of alias relations (`p`, `*a`), (`*p`, `**a`), etc.

Shape analysis has not been found to scale to medium sized programs [13].

This chapter will explore how pointer information is represented and used by compilers, recent research on scaling flow-sensitive algorithms, different approaches to handling field-sensitivity, demand-driven versus exhaustive algorithms, and how dynamic objects are represented in a points-to-set-based pointer analysis.

5.1 Representation

In our abstraction, data dependencies are expressed between expressions that appear in a program. A different approach to representing data dependencies in the presence of indirect memory operations, is to decompose indirect memory operations into their effects on individual variables. A *Static single assignment* (SSA) form is a code representation where each variable is only defined once [5]. Variables are *versioned* such that a redefinition of

```
x = &s;
y = &s;
a = &y;
b = &y;
p = ...
  ? &a
  : &b;

*p = &x;



c = *p;



*c = &t;



z = y;
```
(a) Original code

```
x_0 = &s;
y_0 = &s;
a_0 = &y;
b_0 = &y;
p_0 = ...
  ? &a
  : &b;

*p_0 = &x;
a_1 = χ(a_0);
b_1 = χ(b_0);

c_0 = *p_0;
μ(a_1);
μ(b_1);

*c_0 = &t;
x_1 = χ(x_0);
y_1 = χ(y_0)

z_0 = y_1;
```
(b) Factored SSA form

```
x_0 = &s;
y_0 = &s;
a_0 = &y;
b_0 = &y;
p_0 = ...
  ? &a
  : &b;

(*p_0)_0 = &x;
a_1 = χ(a_0);
b_1 = χ(b_0);

c_0 = (*p_0)_0;
μ(a_1);
μ(b_1);

(*c_0)_0 = &t;
x_1 = χ(x_0);
y_1 = χ(y_0)

z_0 = y_1;
```
(c) Versioned expressions

Figure 5.1: Example of SSA form with indirect memory operations

a variable defines a new version of the variable. To model indirect memory operations that may define a variable, a *factored SSA* form chains possible definitions of variables together [3, 23].

A pointer analysis has to be performed before a program is transformed into factored SSA form, because the occurrences of potential definitions of variables must be known to perform the transformation.

Listing 5.1(b) is a transformation of the code in Listing 5.1(a) into factored SSA form. Versions of variables are expressed as numbers in subscripts. We assume that an ordinary points-to-set-based pointer analysis [8] was performed before generating Listing 5.1(b). Statements involving indirect memory operations are annotated with χ and μ functions that express possible definitions and uses of variables [4]. A χ function indicates a possible definition of a variable: the function takes a version of a variable as an argument and returns a new version of the variable after the potential definition. A μ function indicates a possible use of a version of a variable.

Value numbering is a method used to detect expressions that evaluate to the same value [19]. Chow *et al.* describe a method of performing value numbering of expressions after a program is transformed into factored SSA form [4]. In Chow *et al.*'s *Hashed SSA* algorithm, dereference expressions are also versioned: Listing 5.1(c) is a transformation of Listing 5.1(b) with versioned dereference expressions. The two occurrences of the expression "*p" is determined to be the same version, and the analysis is able to deduce that c_0 must have the value "&x" after the assignment. Although the analysis was able to obtain a precise value for c_0, imprecision arises from the fact that the points-to-set-based pointer analysis did not have the information deduced above when the pointer analysis was performed.

When a points-to-set-based pointer analysis is performed on Listing 5.1(a), the following occurs: On line 9, "p" points to two locations, "a" and "b", and the two variables are weakly updated to point to "x" and "y". On line 13, "a" and "b" are loaded and "c" points to "x" and "y" after the assignment. On line 17, the store to "*c" weakly updates "x" and "y". Thus, in Listing 5.1(c), the code is annotated with a spurious possible definition of "y" on line 19, and "z_0" is deemed to have a value of "&s" or "&t".

A possible way to reduce the loss of precision of using imprecise pointer information for SSA construction is to iterate pointer analysis, SSA transformation, then variable substitution multiple times. The precision of a points-to-set-based pointer analysis may improve after variable substitution, and the resulting SSA form may be more precise than the previous iteration. However, iterating these three steps would be inefficient. In contrast, our algorithm obtains a precise result without preprocessing the input. Figure 5.2 is the EDF graph of Listing 5.1(a). The precise value of "z" is determined by the algorithm.

5.2 Flow-sensitivity

A *sparse* data-flow analysis attempts to propagate data-flow information only to points that use the information. In contrast, a *dense* algorithm propagates all data-flow information to all points of the program. A recent paper has scaled a sparse flow-sensitive algorithm to over a million lines of code [10].

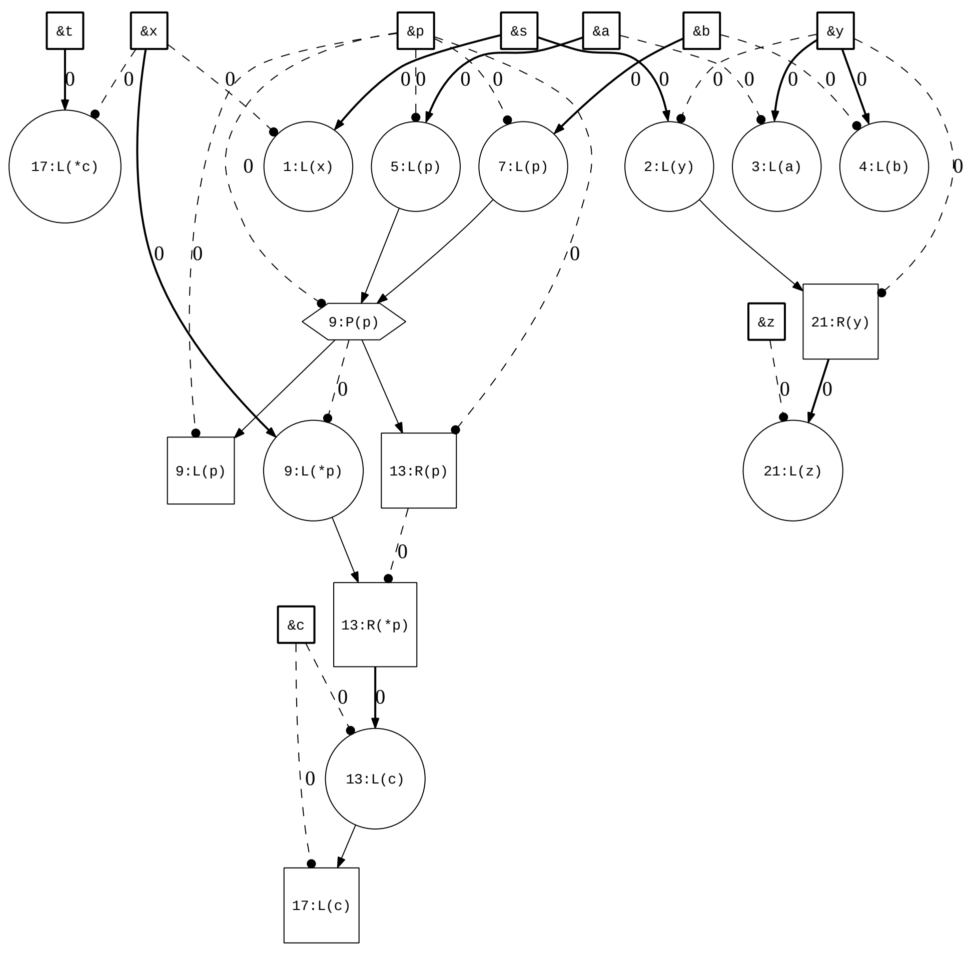


Figure 5.2: Graph of Listing 5.1(a)

SSA form is widely used to implement sparse algorithms for data-flow analyses, such as constant propagation [25]. Hardekopf and Lin used an SSA representation for non-address-taken variables but used a dense algorithm for all address-taken variables [9, 10]. Since non-address-taken variables cannot be indirectly referenced by pointers, definitions and uses of the variables are readily apparent.

Yu *et al.* assigns *points-to levels* to each variable such that a variable that may point to another variable is at a higher or equal level than the pointed-to variable, and the algorithm analyzes variables in decreasing order of their level [30]. In this way, SSA form can be used for variables in higher levels because they cannot be indirectly referenced by variables in lower levels. Staiger-Stöhr presented a method to incrementally build the SSA form of a program without analyzing pointers level-by-level [22].

Although sparse techniques may potentially yield a scalable pointer analysis, there are some problems with its practicality on real-world programs: an unanalyzable statement may potentially modify all address-taken memory locations. This conflicts with sparse techniques that rely on the assumption that each statement uses and modifies a small amount of pointer information. If statements modify a large amount of pointer information, then the performance benefits of a sparse algorithm over a dense one is reduced. If an algorithm is to be practical for production compilers, this issue needs to be resolved.

5.3 Field-sensitivity

One important consideration when designing a static analysis for C is its weak-typing. Aside from the problem of type casts to pointer types, handling fields is another problem. Pearce *et al.* present an algorithm that is field-sensitive [17]. Their goal is to model *portable* C programs only, which places restrictions on the use of casts and pointer arithmetic. Papers, including a Ph.D. thesis, have been devoted to designing a pointer analysis that is sound for programs that use unsafe or non-portable language features of C [16, 24].

Wilson and Lam's analysis forgoes utilization of unreliable type information and represents memory locations using location sets [29]. Such an abstraction can soundly handle arbitrary pointer arithmetic. Their analysis relies on the ability to differentiate between array accesses and field accesses [28]. This ability is important because pointer arithmetic inside a loop may generate an unbounded number of offsets within an object. Our algorithm does not require this ability.

5.4 Demand-driven Analysis

Few papers have described a demand-driven pointer analysis. Hientze and Tardieu present deduction rules for computing only a subset of points-to pairs that are relevant to answering a particular points-to query [11]. The algorithm is flow-insensitive and context-insensitive.

Thomas Reps presents how a number of program analysis can be formulated as a *context-free language reachability* problem, which is an extension of the graph reachability problem where paths are restricted by a context-free language [18]. For example, in a context-sensitive data-flow problem, data-flow facts that enter a procedure from a particular call-site should only leave the procedure through the same call-site. If we view the trace of call-site entries and exits of a data-flow fact as a string, then the context-sensitivity requirement can be formulated as a context-free language that requires that the entries and exits be balanced. Sridharan *et al.* apply this technique to create a demand-driven pointer analysis algorithm for Java [21]. Zheng and Rugina use a similar technique for a flow-insensitive analysis for C [31].

5.5 Dynamic Object Representation

A points-to analysis abstracts dynamic objects by categorizing them in some way. The most imprecise method is to use a single abstract object to represent all dynamic objects [8]. A common alternative is to categorize dynamic objects by their allocation sites.

Dynamic objects can be distinguished by varying levels of context-sensitivity. A call-string approach to context-sensitivity is to use the chain of call sites on the call stack to differentiate calls to a single procedure. Then, dynamic objects allocated at the same site are differentiated by call-strings [14].

A problem with differentiating dynamic objects by call-strings is that the number of call-strings is exponential with respect to the depth of a call-graph. The number of call-strings can be limited by limiting the length of call-strings to a fixed number. Even if call-strings are limited to a length of two, the average number of contexts for each procedure may be in the hundreds [15].

A *Binary Decision Diagram* (BDD) is a data structure that can compactly represent a large set. BDDs have been used to represent points-to sets [1], and can also be extended to compactly represent context-sensitive pointer information [34, 33, 15, 27]. Most BDD-based analyses are flow-insensitive, but they obtain some of the benefits of flow-sensitivity by transforming input programs into SSA form. Zhu presents a flow- and context- sensitive pointer analysis using BDDs, but due to limitations on the operations that can be performed on BDDs, the analysis is unable to perform indirect strong updates [32].

This concludes the related work chapter. The final chapter is the conclusion of this thesis.

Chapter 6

Conclusion

Pointer analysis is a prerequisite to many static analyses and compiler optimizations. As the sophistication of compiler optimizations increases, accurate pointer information becomes increasingly important.

The major advantage of performing a flow-sensitive pointer analysis is the ability to perform strong updates. Most points-to-set-based analyses perform strong updates on a limited subset of memory locations: memory locations associated with local variables in a non-recursive procedure. Strong updates are not performed on memory locations outside of this subset. The lack of strong updates on non-singular objects has the effect that, as the analysis of a program progresses, non-singular objects "accumulate" possible values, because old values are never removed. The occurrence of redefinitions of memory locations may not be frequent. However, the lack of strong updates on non-singular objects is troublesome if an unanalyzable construct appears in a program: once a non-singular object is assumed to have any value at some point in a program, it will stay in that state for the rest of the program.

In this thesis, we presented a flow-sensitive algorithm that treats all memory locations equally with respect to strong updates: if the point in a program that stored last to a memory location can be determined, stores to the memory location that happened earlier are not reaching definitions to loads of the memory location. Unanalyzable constructs are handled such that if obtaining precise pointer information is not possible, the algorithm performs an over-approximation so that the analysis may complete quickly. Field-sensitivity is enabled without relying on type information, which enables sound analysis of unportable or unsafe

C constructs. The representation computed by the algorithm consumes little memory, and preliminary evaluations indicate that the analysis can be performed swiftly.

6.1 Future Work

A logical continuation of our research is to transform our analysis into an interprocedural analysis. A possible approach is the *super-graph* approach to interprocedural analysis that treats procedural calls and returns as ordinary intraprocedural jumps. The result is a single massive control flow graph of the entire program.

A super-graph approach loses precision due to *impossible paths*. An informal description of impossible paths is that pointer information that enters a procedure is propagated to all return-sites of the procedure, not just the return-site of the caller of the procedure.

If a program uses function pointers, then the full call-graph of the program is unknown until pointer analysis is performed. Many pointer analyses encounter this obstacle, and a common solution is to incrementally build the call-graph during pointer analysis.

One complication of incrementally constructing a super-graph is the maintenance of various control flow graph related data structures. Our algorithm uses a dominator tree to answer dominance queries and a dominance frontier mapping to place ϕ nodes. Both data structures have to be updated when a super-graph grows.

The size of the dominance frontier mapping has empirically been shown to vary linearly with program sizes [5]. However, the size characteristic of the dominance frontier mapping for super-graphs is unknown.

Bibliography

[1] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 103–114, New York, NY, USA, 2003. ACM.

[2] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 296–310, New York, NY, USA, 1990. ACM.

[3] Jong-Deok Choi, R. Cytron, and J. Ferrante. On the efficient engineering of ambitious program analysis. *IEEE Trans. Softw. Eng.*, 20:105–114, February 1994.

[4] Fred C. Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. Effective representation of aliases and indirect memory operations in SSA form. In *Proceedings of the 6th International Conference on Compiler Construction*, pages 253–267, London, UK, 1996. Springer-Verlag.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13:451–490, October 1991.

[6] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 230–241, New York, NY, USA, 1994. ACM.

[7] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 365–372, New York, NY, USA, 1987. ACM.

[8] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 242–256, New York, NY, USA, 1994. ACM.

[9] Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *POPL '09: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 226–238, New York, NY, USA, 2009. ACM.

[10] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, pages 289 –298, April 2011.

[11] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 24–34, New York, NY, USA, 2001. ACM.

[12] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM.

[13] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, 1999.

[14] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 278–289, New York, NY, USA, 2007. ACM.

[15] Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18:3:1–3:53, October 2008.

[16] Antoine Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems*, LCTES '06, pages 54–63, New York, NY, USA, 2006. ACM.

[17] David J. Pearce, Paul H.J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of C. *ACM Trans. Program. Lang. Syst.*, 30(1):4, 2007.

[18] Thomas Reps. Program analysis via graph reachability. In *ILPS '97: Proceedings of the 1997 International Symposium on Logic Programming*, pages 5–19, Cambridge, MA, USA, 1997. MIT Press.

[19] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–27, 1988.

[20] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 16–31, New York, NY, USA, 1996. ACM.

[21] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for Java. In *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 59–76, New York, NY, USA, 2005. ACM.

[22] Stefan Staiger-Stöhr. Implementing sparse flow-sensitive Andersen analysis. Technical Report 3, Institute of Software Technology, University of Stuttgart, 2009.

[23] Bjarne Steensgaard. Sparse functional stores for imperative programs. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, IR '95, pages 62–70, New York, NY, USA, 1995. ACM.

[24] Ning Wang. *From assumptions to assertions: A sound and precise points-to analysis for the C language*. PhD thesis, Irvine, CA, USA, 2007. AAI3282825.

[25] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991.

[26] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[27] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM.

[28] Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 1–12, New York, NY, USA, 1995. ACM.

[29] Robert Paul Wilson. *Efficient, context-sensitive pointer analysis for C programs*. PhD thesis, Stanford, CA, USA, 1998. AAI9837324.

[30] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *CGO '10: Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 218–229, New York, NY, USA, 2010. ACM.

[31] Xin Zheng and Radu Rugina. Demand-driven alias analysis for C. In *POPL '08: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 197–208, New York, NY, USA, 2008. ACM.

[32] Jianwen Zhu. Towards scalable flow and context sensitive pointer analysis. In *Proceedings of the 42nd Annual Design Automation Conference*, DAC '05, pages 831–836, New York, NY, USA, 2005. ACM.

[33] Jianwen Zhu and S. Calman. Context sensitive symbolic pointer analysis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(4):516 – 531, April 2005.

[34] Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 145–157, New York, NY, USA, 2004. ACM.